SYSTEM SPECIFICATION AND VERIFICATION

Why Specify?

The purpose of writing a specification is to prevent errors

- Specification is not an end in itself
- Engineers should be able to apply specification techniques as appropriate

- Writing a specification can guide the design process
  - Having to describe a design precisely often reveals problems
  - Start early!
    - As system ideas are being considered
    - Spec will be incomplete and incorrect

- Specifications can provide a clear, concise way of communicating design intent
  - Guide to implement and test a system

Writing a spec helps you think clearly. Thinking clearly is hard!

Specification Preliminaries

abstraction

- Specification is an abstraction of a system
  - Described as an algorithm or building blueprints
  - Distinction between hardware and software blurred
  - Specifying first helps think about problem before writing the code

- Levels of formalism/verification
  - Back-of-the-envelope
  - Writing prose
  - Formal specification
  - Formal verification

  Ricky Butler, NASA, experience/gut-feel: "80% boost by just b-o-t-e, then diminishing returns"

- Smaller/simpler is better
  - Short spec easier to understand
    - e.g 10 lines rather than 10,000
  - Debugging actual code not as efficient as debugging spec
    - specification process guides development
  - Easier to evaluate future changes
  - Easier to evaluate potential options <- on the fly!

Specification Preliminaries

TLA+

- We'll focus on distributed systems
  - communication between collaborative agents using a protocol

- TLA+: Temporal Logic of Actions
  - A TLA+ specification is a formal description
  - Not a programming language---specs aren't executed
  - Specifications can be model checked
    - determine if all possible behaviors of system satisfy desired properties
  - Stylized (mathematical) way of writing abstract definitions

- PlusCal language
  - write spec as algorithm that can be translated into a TLA+ spec (and then checked by TLA+ tools)

TLA+

Leslie B. Lamport … is best known for his seminal work in distributed systems and as the initial developer of the document preparation system LaTeX.[2] Leslie Lamport was the winner of the 2013 Turing Award[3] for imposing clear, well-defined coherence on the seemingly chaotic behavior of distributed computing systems, in which several autonomous computers communicate with each other by passing messages. He devised important algorithms and developed formal modeling and verification protocols that improve the quality of real distributed systems. These contributions have resulted in improved correctness, performance, and reliability of computer systems. [Wikipedia]

- Read
  - Lamport paper: "Use of Formal Methods at Amazon Web Services"
  - Lamport, Specifying Systems Chapter 1-3
- Complete Principles and Specification Track Tutorial sections covering:
  - Open a new spec
    - Add simple 1 bit clock: Init1 and Next1
  - Saving a module causes the toolbox to parse it.
    - See the errors, add declaration for variable b
  - Try the pretty printer
  - Create a new model to check the design
    - Identify Init1 and Next1 and run TLC
    - Replace 0 in def with string and rerun TLC
    - Observe statistics: diameter
  - Add type invariant property
    - On the model overview page and rerun TLC
    - As a definition in the spec.
  - Another language: PlusCal

Propositional Logic

Unary operations

in	Result			
F	NOT	NOP	ONE	ZERO
0	1	0	1	0
1	0	1	1	0

*Convenient to use 1 and 0 in place of *True* and *False*

Propositional Logic

Unary operations

in	Result			
F	NOT	NOP	ONE	ZERO
0	1	0	1	0
1	0	1	1	0

¬F

Binary operations

input		Result							
F	G	OR	AND	XL	L	XR	R	XOR	ZERO
0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	1	1	1	0
1	0	1	0	1	1	0	0	1	0
1	1	1	1	0	1	0	1	0	0

F ᵛ G
F ^ G

input		Result							
F	G	NOR	NXOR	NR	NXR	NL	NXL	NAND	ONE
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1
			EQ				IMP		

F ≡ G
F => G

(F=>G) ≡¬FᵛG

Basic Boolean Laws

Commutative Laws	$x*y = y*x$	$x+y = y+x$
Associative Laws	$x*(y*z) = (x*y)*z$	$x+(y+z) = (x+y)+z$
Distributive Laws	$x*(y+z) = x*y + x*z$	$x*(y+z) = x*y + x*z$
Inverse Laws	$x * \bar{x} = 0$	$x + \bar{x} = 1$
Identity Laws	$x * 1 = x$	$x + 0 = x$
	$x * x = x$	$x + x = x$
	$x * 0 = 0$	$x + 1 = 1$
DeMorgan's Law	$x * y = \overline{\bar{x} + \bar{y}}$	$x + y = \overline{\bar{x} * \bar{y}}$
Double Negation Law	$\bar{\bar{x}} = x$	

* operator is conjunction/AND
+ operator is disjunction/OR

Sets

$\cap$	Intersection	
$\cup$	Union	UNION{ {1,2}, {2,3}, {3,4} } = {1,2,3,4}
$\subseteq$	Subset	SUBSET {1,2} = { {}, {1}, {2}, {1,2} }
$\setminus$	Set difference	({1,-1/2} $\cup$ {1,3,4}) $\setminus$ ({1,5,7} $\cap$ {1,4,5,7}) = {-1/2, 3, 4}
$\in$	Is an element of	

Cardinality(S)	the number of elements in set S, if S is finite
IsFiniteSet(S)	True iff S is a finite set

Predicate Logic

$\forall$	Universal Quantification (for all)	$\forall x \in$ int : (x >= 0)
$\exists$	Existential Quantification (there exists)	$\exists y \in$ int : (y >8)

Formulas and Language

2*x > x is this a formula or a statement of fact?

TLA+ is untyped

- Lamport:
  For programming languages the benefits [of types] seem to outweigh the costs. For writing specifications, I have found the costs outweigh the benefits.

  x == 18
  x == “leaf”
  x == T
  x == 3.14
  x == <1,2,5>

- Every syntactically well formed expression has a meaning
- We will need to add constraints that restrict what values a variable may take

Ch 2: Specifying a simple clock

System Behavior: a sequence of states

- State: assignment of values to variables
- Function from time to state:

  [hr = 11] → [hr = 12] → [hr = 1] → [hr = 2] → …

  - *F(0)* is the initial condition

An Hour Clock Specification

- Define the state: $hr \in \{1, \ldots, 12\}$

  $HCini \triangleq hr \in \{1, \ldots, 12\}$
- Define the next-state relation (*action*)

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

Note ' notation for next state

Temporal Operators

- □ after each future step (*box)*
- ◇ eventually

First Specification

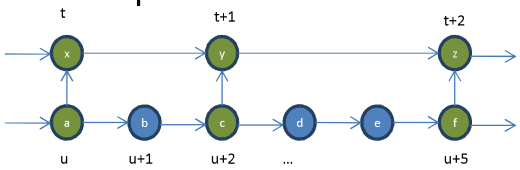
$$HCini \wedge \Box HCnxt$$

Building a larger system

- Multiple variables and multiple clocks

  [hr=11, tmp=23.5] → [hr=12, tmp=23.5] → [hr=12, tmp=23.4] → [hr=12, tmp=23.3] → [hr= 1, tmp=23.5] → …

  - □*HCnxt* requires *hr* to change every step
  - Another example



- Stuttering steps

  $[HCnxt]_{hr} \triangleq HCnxt \vee (hr' = hr)$

- Revised Specification

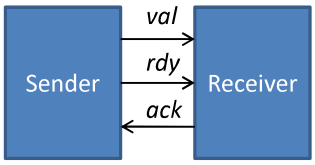
  $HCini \wedge \Box [HCnxt]_{hr}$

TLA Specification

```
------------------ MODULE Hour Clock -----------------
EXTENDS Naturals
VARIABLE hr

HCini == hr \in (1 .. 12)

HCnxt == hr' = IF hr # 12 THEN hr+1 ELSE 1

HC == HCini /\ [][HCnxt]_hr
-----------------------------------------------------------------
THEOREM HC => []HCini
=================================
```

Ch3: An Asynchronous Interface

- Two-phase handshake protocol



- Model at RTL level
- Model for any type of *val*
- Definitions:
  - *TypeInvariant*
  - Initial state
  - Actions: send and receive
  - Next action
  - Complete specification

```
------------------ MODULE AsynchInterface ---------------------
EXTENDS Naturals
CONSTANT  Data
VARIABLES val, rdy, ack

TypeInvariant == /\ val \in Data
                 /\ rdy \in {0, 1}
                 /\ ack \in {0, 1}
-----------------------------------------------------------------
Init == /\ val \in Data
        /\ rdy \in {0, 1}
        /\ ack = rdy

Send == /\ rdy = ack
        /\ val' \in Data
        /\ rdy' = 1 - rdy
        /\ UNCHANGED ack

Rcv  == /\ rdy # ack
        /\ ack' = 1 - ack
        /\ UNCHANGED <<val, rdy>>

Next == Send \/ Rcv

Spec == Init /\ [][Next]_<<val, rdy, ack>>
-----------------------------------------------------------------
THEOREM Spec => []TypeInvariant
=================================================================
```

Conventions

```
Send ==  /\ rdy = ack
         /\ val' \in Data
         /\ rdy' = 1 - rdy
         /\ UNCHANGED ack
```

Guard first

What conditions must be true for the action to *potentially* occur?

- State using only current variable values

Future variable values

Next variable value as relates to current values

Variables that don't change

Choice

Next == Send \/ Rcv

If "guards" for Send and Rcv are TRUE, either option is acceptable

The definitions of Send and Rcv include guards whose conditions are mutually exclusive:

```
Send ==   /\ rdy = ack  ...
Rcv  ==   /\ rdy # ack  ...
```

Other examples:

```
/\ val' \in Data

/\ rdy \in {0, 1}
```
← *note the set notation using { }*

```
HCini == hr \in (1 .. 12)
```
← *note the set creation using ..*

```
CoinFlip == \/ coin' = "heads"
            \/ coin' = "tails"
```

Two-phase handshake spec improvements

Package up functionality as a reusable module: Channel

- Data structure: replace 3 variables (*rdy*, *val*, and *ack*) with 1 variable
  - as an ordered triple: *chan = < -1/2, 1, 0 >*
  - or better as a record: *chan = [val:Data, rdy:{0,1}, ack:{0,1}]*
- Characterize sending a specific value

Send(d) ==
/\ chan.rdy = chan.ack
/\ chan' = [val |-> d, rdy |-> 1-chan.rdy, ack |-> chan.ack]

Prev definition
Send == /\ rdy = ack
/\ val' \in Data
/\ rdy' = 1 - rdy
/\ UNCHANGED ack

Send(d) ==
/\chan.rdy = chan.ack
/\ chan' = [chan EXCEPT !.val=d, !.rdy=1-@]

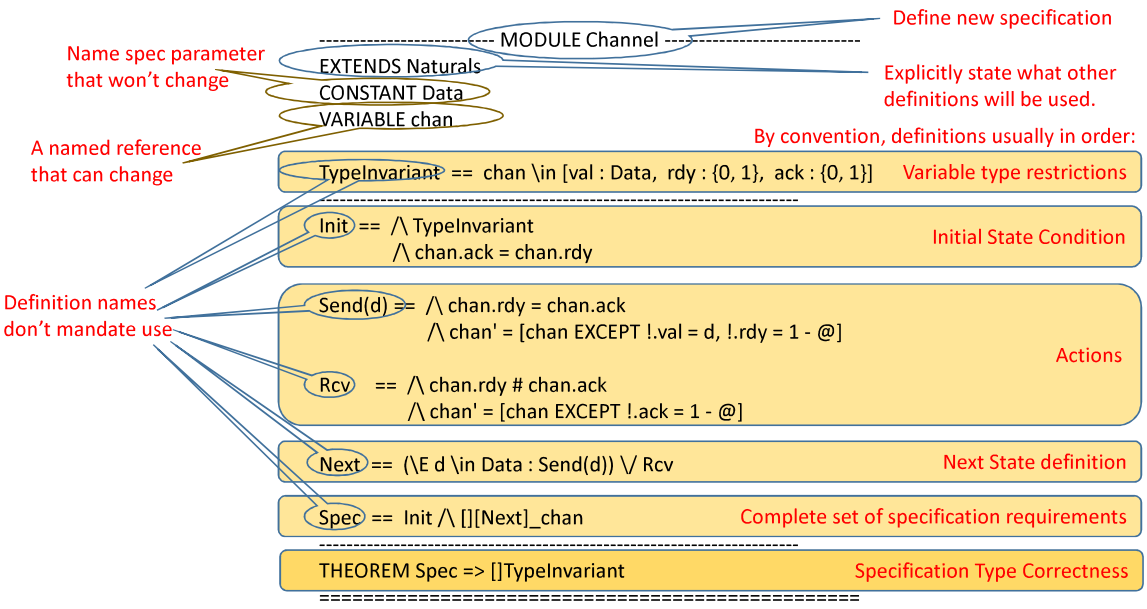
Next == (\E d \in Data: Send(d)) \/ Rcv

Prev definition
Next == Send \/ Rcv

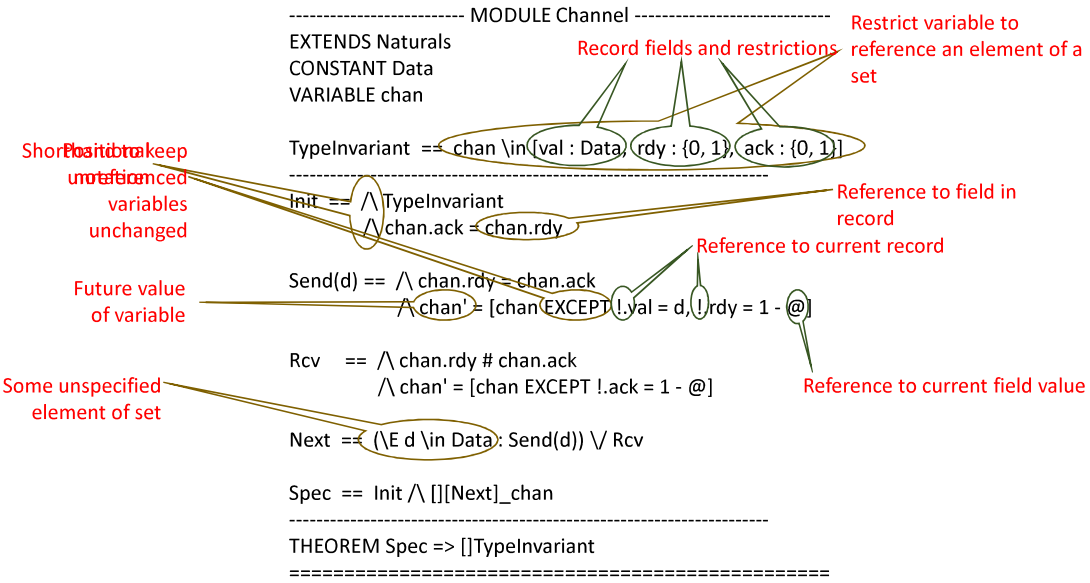
New two-phase handshake spec

```
---------------------------- MODULE Channel ------------------------------
EXTENDS Naturals
CONSTANT Data
VARIABLE chan

TypeInvariant  ==  chan \in [val : Data,  rdy : {0, 1},  ack : {0, 1}]
--------------------------------------------------------------------------
Init  ==  /\ TypeInvariant
          /\ chan.ack = chan.rdy

Send(d) ==  /\ chan.rdy = chan.ack
            /\ chan' = [chan EXCEPT !.val = d, !.rdy = 1 - @]

Rcv     ==  /\ chan.rdy # chan.ack
            /\ chan' = [chan EXCEPT !.ack = 1 - @]

Next  ==  (\E d \in Data : Send(d)) \/ Rcv

Spec  ==  Init /\ [][Next]_chan
--------------------------------------------------------------------------
THEOREM Spec => []TypeInvariant
==========================================================
```

Two-phase handshake spec



Two-phase handshake spec



Original Specification

```
---------- MODULE AsynchInterface ----------
EXTENDS Naturals
CONSTANT  Data
VARIABLES val, rdy, ack

TypeInvariant == /\ val \in Data
                 /\ rdy \in {0, 1}
                 /\ ack \in {0, 1}
-------------------------------------------------------------
Init == /\ val \in Data
        /\ rdy \in {0, 1}
        /\ ack = rdy

Send == /\ rdy = ack
        /\ val' \in Data
        /\ rdy' = 1 - rdy
        /\ UNCHANGED ack

Rcv  == /\ rdy # ack
        /\ ack' = 1 - ack
        /\ UNCHANGED <<val, rdy>>

Next == Send \/ Rcv

Spec == Init /\ [][Next]_<<val, rdy, ack>>
-------------------------------------------------------------
THEOREM Spec => []TypeInvariant
=====================================
```

New Specification

```
---------- MODULE Channel ----------
EXTENDS Naturals
CONSTANT Data
VARIABLE chan

TypeInvariant  ==  chan \in [val : Data,  rdy : {0, 1},  ack : {0, 1}]
-------------------------------------------------------------------
Init  ==  /\ TypeInvariant
          /\ chan.ack = chan.rdy

Send(d) ==  /\ chan.rdy = chan.ack
            /\ chan' = [chan EXCEPT !.val = d, !.rdy = 1 - @]

Rcv     ==  /\ chan.rdy # chan.ack
            /\ chan' = [chan EXCEPT !.ack = 1 - @]

Next  ==  (\E d \in Data : Send(d)) \/ Rcv

Spec  ==  Init /\ [][Next]_chan
-------------------------------------------------------------------
THEOREM Spec => []TypeInvariant
===================================================
```

Ch 4: A FIFO

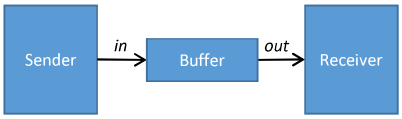
New built-in module: *Sequences*

Seq(set) — <3, 2, 7> is an element of Seq(Nat)

Operations:

Head(s)	Head(<1,2,3>) = 1
Tail(s)	Tail(<1,2,3>) =3
Append(s,e)	Append(<1,2,3>, 4) = <1,2,3,4>
s o t	<1,2,4> o <3,5> = <1,2,4,3,5>
Len(s)	Len(<1,4,5>) = 3

Module Instantiation



- Channel
  Data structure: [val : Data, rdy : {0, 1}, ack : {0, 1}]
  Operations: Send(d), Rcv

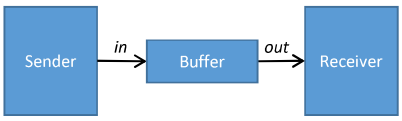
- Instantiation of Channel
  myChan == INSTANCE Channel WITH Data <- Message, chan <- in

- Operations
  myChan!Send(msg)

  myChan!Rcv

```
---------------------- MODULE Channel ----------------------
EXTENDS Naturals
CONSTANT Data
VARIABLE chan

TypeInvariant == chan \in [val : Data, rdy : {0, 1}, ack : {0, 1}]
------------------------------------------------------------
Init == /\ TypeInvariant
        /\ chan.ack = chan.rdy

Send(d) == /\ chan.rdy = chan.ack
           /\ chan' = [chan EXCEPT !.val = d, !.rdy = 1 - @]

Rcv  == /\ chan.rdy # chan.ack
        /\ chan' = [chan EXCEPT !.ack = 1 - @]

Next == (\E d \in Data : Send(d)) \/ Rcv

Spec == Init /\ [][Next]_chan
------------------------------------------------------------
THEOREM Spec => []TypeInvariant
============================================================
```

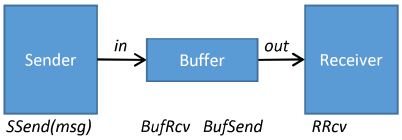
FIFO Specification



SSend(msg), BufRcv *BufSend, RRcv*

- Definitions for:
  - Instantiations of Channel
    - In-Channel
    - Out-Channel
  - Type invariant
    - channel invariants
    - message queue invariant
  - Initial state
  - Actions: send/receive per channel
    - Reuse of channel operations with message queue update operations
  - Next action
  - Complete specification

InnerFIFO Specification




```
---------------------------- 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 \in Seq(Message)
```

```
SSend(msg)  ==  /\ InChan!Send(msg)          Send msg on channel in
                /\ UNCHANGED <<out, q>>

BufRcv == /\ InChan!Rcv                      Receive message from channel in
          /\ q' = Append(q, in.val)          and append it to tail of q
          /\ UNCHANGED out

BufSend == /\ q # << >>                      Enabled only if q is nonempty
           /\ OutChan!Send(Head(q))          Send Head(q) on channel out
           /\ q' = Tail(q)                   and remove it from q
           /\ UNCHANGED in

RRcv ==  /\ OutChan!Rcv                      Receive message from channel out
         /\ UNCHANGED <<in, q>>

Next == \/ \E msg \in Message : SSend(msg)
        \/ BufRcv
        \/ BufSend
        \/ RRcv

Spec == Init /\ [][Next]_<<in, out, q>>
------------------------------------------------------------------------------
THEOREM Spec => []TypeInvariant
==============================================================
```

Hiding the Queue

- The queue is an internal variable and should be hidden in the final specification
- Use existential quantifier

```
------------------------- MODULE FIFO --------------------------
CONSTANT Message
VARIABLES in, out
Inner(q) == INSTANCE InnerFIFO WITH q<-q, in<-in, out<-out, Message <-Message
Spec == \EE q : Inner(q)!Spec
==============================================================
```

A Bounded FIFO

- Disable receiving messages when FIFO is full
  - Add constraint to *BufRcv:* *Len(q) < N*
- Create new Module using InnerFIFO

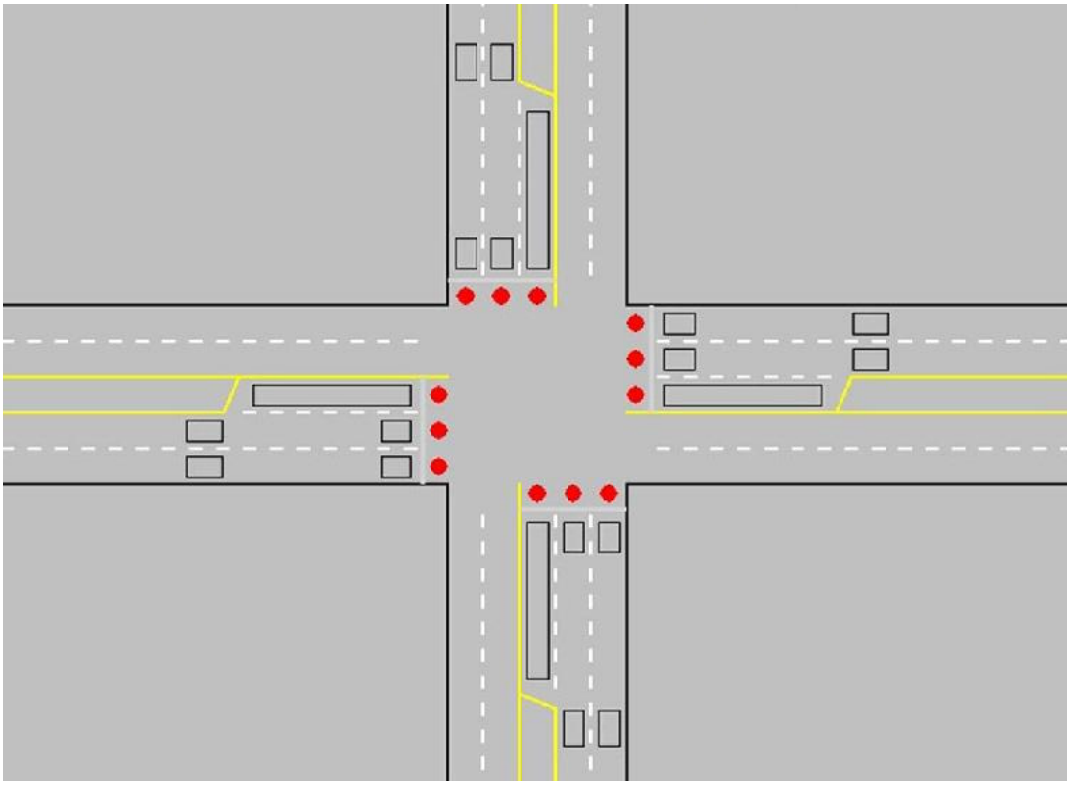
```
---------------------- MODULE BoundedFIFO -------------------------
EXTENDS Naturals, Sequences
VARIABLES in, out
CONSTANT Message, N
ASSUME ( N \in Nat) /\ (N>0)

Inner(q) == INSTANCE InnerFIFO

Bnext(q) == /\ Inner(q)!Next
            /\ Inner(q)!BufRcv => (Len(q) < N)

Spec == \E q : Inner(q)!Init /\ [][Bnext(q)]_<<in,out,q>>
==============================================
```

Closed-system Specification

- *FIFO* describes the buffer, the sender, and receiver
- May wish to explicitly define system and environment actions separately

```
Next == SysNext \/ EnvNext

SysNext == BufRcv \/ BufSend
EnvNext == \E msg \in Message : SSend(msg)) \/ RRcv
```

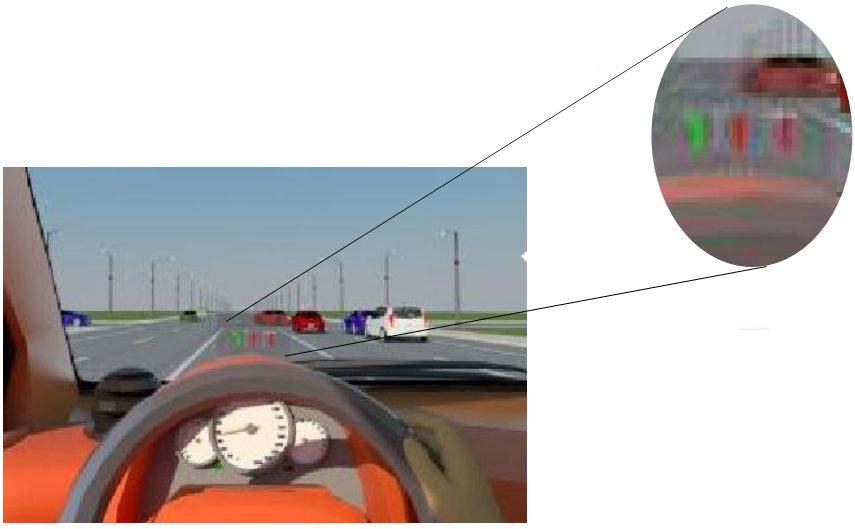
Consider a traffic light…..



Virtual Traffic Light Project

http://www.cnn.com/2015/01/14/tech/virtual-traffic-lights-windshield/index.html

Researchers at Carnegie Mellon University claim they can reduce the commute times of urban workers by 40% by replacing physical traffic lights with virtual traffic lights.



http://users.ece.cmu.edu/~tonguz/vtl/about.html

```
--------------------------- MODULE traffic_light1 ---------------------------
(*
   This first example considers an intersection where traffic is moving in
   2 directions---either North-South (Dir1) or East-West (Dir2)

   Variables
        light   a two-tuple with the first value the color of the Dir1 lights and
                the second value the color of the Dir2 lights.

        state   an index into a sequence of safe values (LightSequence)

   To avoid accidents, lights will change only in the order defined in LightSequence
*)
EXTENDS Naturals

VARIABLE state, light

Dir1Green  == <<"green", "red">>
Dir1Yellow == <<"yellow","red">>
Dir2Green  == <<"red",   "green">>
Dir2Yellow == <<"red",   "yellow">>
AllStop    == <<"red",   "red">>

LightSequence == <<Dir1Green, Dir1Yellow, AllStop, Dir2Green, Dir2Yellow, AllStop>>
LightSequenceLen == 6

TypeInvariant == /\ light \in {Dir1Green, Dir1Yellow, Dir2Green, Dir2Yellow, AllStop}
                 /\ state \in (1..LightSequenceLen)

Init == /\ light = Dir1Green
        /\ state = 1

Next == LET nextState == IF state # LightSequenceLen THEN state+1 ELSE 1
        IN   /\ state' = nextState
             /\ light' = LightSequence[nextState]

Spec == Init /\ [][Next]_<<state, light>>

-------------------------------------------------------------------------------

THEOREM Spec => []TypeInvariant

===============================================================================
```

MODULE *traffic_light1*

This first example considers an intersection where traffic is moving in 2 directions— either North-South (*Dir*1) or East-West (*Dir*2)

Variables light a two-tuple with the first value the color of the *Dir*1 lights and the second value the color of the *Dir*2 lights.

state an index into a sequence of safe values (*LightSequence*)

To avoid accidents, lights will change only in the order defined in *LightSequence*

EXTENDS *Naturals*

VARIABLE *state*, *light*

$$
\begin{aligned}
Dir1Green &\triangleq \langle \text{"green"}, \text{"red"} \rangle \\
Dir1Yellow &\triangleq \langle \text{"yellow"}, \text{"red"} \rangle \\
Dir2Green &\triangleq \langle \text{"red"}, \text{"green"} \rangle \\
Dir2Yellow &\triangleq \langle \text{"red"}, \text{"yellow"} \rangle \\
AllStop &\triangleq \langle \text{"red"}, \text{"red"} \rangle
\end{aligned}
$$

$$LightSequence \triangleq \langle Dir1Green, Dir1Yellow, AllStop, Dir2Green, Dir2Yellow, AllSto$$

$$LightSequenceLen \triangleq 6$$

$$
\begin{aligned}
TypeInvariant \triangleq\ & \land light \in \{Dir1Green, Dir1Yellow, Dir2Green, Dir2Yellow, AllS \\
& \land state \in (1 \,..\, LightSequenceLen)
\end{aligned}
$$

$$
\begin{aligned}
Init \triangleq\ & \land light = Dir1Green \\
& \land state = 1
\end{aligned}
$$

$$
\begin{aligned}
Next \triangleq\ & \text{LET } nextState \triangleq \text{IF } state \neq LightSequenceLen \text{ THEN } state + 1 \text{ ELSE } 1 \\
& \text{IN} \quad \land state' = nextState \\
& \qquad \land light' = LightSequence[nextState]
\end{aligned}
$$

$$Spec \triangleq Init \land \Box[Next]_{\langle state,\ light \rangle}$$

THEOREM $Spec \Rightarrow \Box TypeInvariant$