TLA+ specification language

- Literal objects: booleans, numbers, strings
- Collections of objects
  - Sets { }
  - Tuples < >
  - Records [ ]
- Unrestricted variables
  - present and future value
- Predicates/expressions
  - relations: /\ \/ = # if-then-else
- Named definitions
  - parameterized named definitions
  - scoped [parameterized] named definitions (LET)
- Modules
  - extended modules
    - add new definitions
    - add new restrictions
  - parameterized modules (CONSTANTS)
  - instantiated modules
  - available modules: naturals, sequences
- Temporal formulas
  - [] <>
  - Stuttering steps
- uses for definitions
  - type invariant
  - initial condition
  - action guard, next variable values
  - nextstate choices bet actions
  - complete spec initial condition+nextstate+…
- Information hiding with existential quantification
- EXAMPLES
  - HourClock
  - AsyncInterface
  - FIFO
  - BoundedBuffer

Records and Tuples are actually Functions!

- Given an argument (field name or index) returns a value
- LightSequence[x] is syntactic sugar for LightSequence(x), a function

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

$$\boxed{\wedge light' = LightSequence[nextState]}$$

- Functions can represent memory and other data structures
  - Memory
    - Read operation notationally expressed as: *mem[x]*
    - Write operation returns a new function: *mem EXCEPT mem[adr] = newVal*
  - Records: function whose domain is a finite set of strings
    - Channel example:
      record domain: {“val”, “ack”, “rdy”}
      r.ack abbreviation for r[“ack”]
  - Two dimensional array is a function that returns functions

2-Way Traffic Light State Representations

- As a tuple (first and second traffic light spec examples):

```
<<"green", "red">>
```

- As a record:

```
lights = [ NS: {"red", "green"}, EW: {"red", "green"} ]
```

- As explicit functions:
  - Let's keep the directions as an index to simplify things a bit

```
TypeInvariant == /\ lights \in [{0,1} -> {"red", "green"}]
```

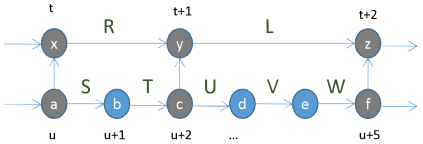
  - Initial state of the system:

```
InitLightFtn(p) == IF p=0 THEN "green" ELSE "red"

Init == /\ lights = [p \in {0,1} |-> InitLightFtn(p)]
```

Specification Writing Advice

- Focus on spec parts most likely to reveal errors
  - TLA+ great for revealing concurrency errors
- Specify a particular view, not the whole system.
  - Must choose what to model
  - May add abstract interface that doesn't actually exist
- Atomicity abstraction critical
  - What can happen in a single step?
    - Coarser-grained simpler, finer-grained more accurate



Label transitions with actions

- Data Structures: keep abstract
  - Will precise layout spec help prevent bugs?
  - Introduce CONTANT parameters

Traffic Light Examples

traffic_light1	sequencing of pairs of light colors for two directions	TLA 1
traffic_light2	added a timer for green light interval	TLA 2
traffic_light3	a new abstraction using a function for light colors dropped yellow lights and the timer to simplify verifying the new abstraction added a NoAccident invariant that finds a bug!	TLA 3
traffic_light4	modified previous to fix problem with a guard	TLA 4
traffic_light5	add back in the timer and yellow lights demo use of PrintT function	TLA 5
traffic_light6	added priority variable so green strictly alternates between directions	TLA 6
traffic_light7	changed setup of initial function so all directions are red	TLA 7
traffic_light8	modify spec to allow for any number of directions (see comments)	TLA 8

Writing the Spec

0. Choose part of system and abstraction
1. Pick variables, define invariant, initial predicate
2. Write next-state action (bulk of work)
   - Sketch out sample behaviors
   - Decompose into disjunction of actions
   - Write compactly and easy to read
   - Take advantage of existing modules/EXTENDS
   - Utilize constant operations for data structure
3. Write temporal part of spec
   - Define liveness, fairness conditions
   - Combine initial predicate, next state action, conditions into def with a single temporal formula
4. Assert theorems about the spec
   - Including type-correctness theorem

Further Hints

- Don't be too clever. Emphasize readability
- Defining a definition with the name "typeInvariant" has no special significance
  - It is not implicitly an assumption
- Don't be too abstract
  - Be sure can capture important events
- Don't assume values that look different are unequal
- Move quantification to the outside
- Prime only what you mean to prime
- Write comments as comments