TLA+ CHECKING

Definitions

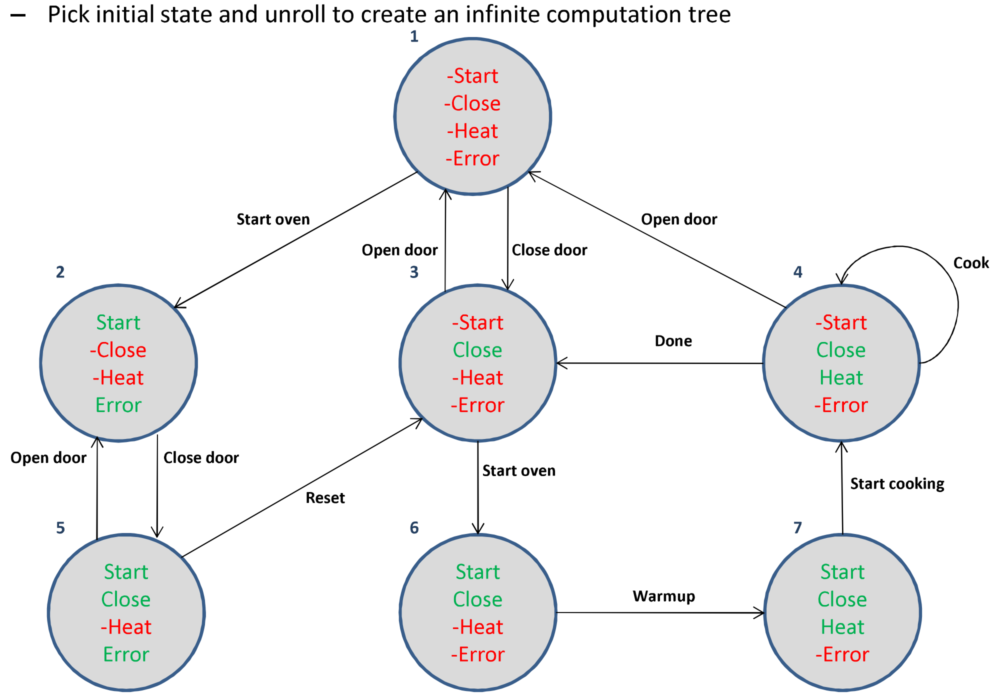
- **State**: mapping of variable names to values *name->value*
- **Behavior**: mapping of time to state *time->(name->value)*
- **Safety property**: assertion of behaviors that should not occur
  - A system where the clock never ticks satisfies any safety properties
- **Liveness property**: assertion of behavior that must occur
  - Properties that must hold for all time --- expressed as temporal formulas

- **Complete Specification:**

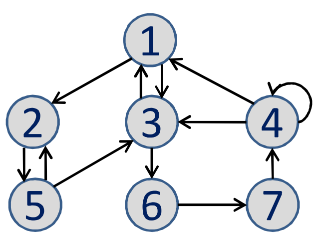
Init	condition constrains the initial state
Next	constrains what steps may occur
Liveness	describe what must eventually happen

Microwave Example

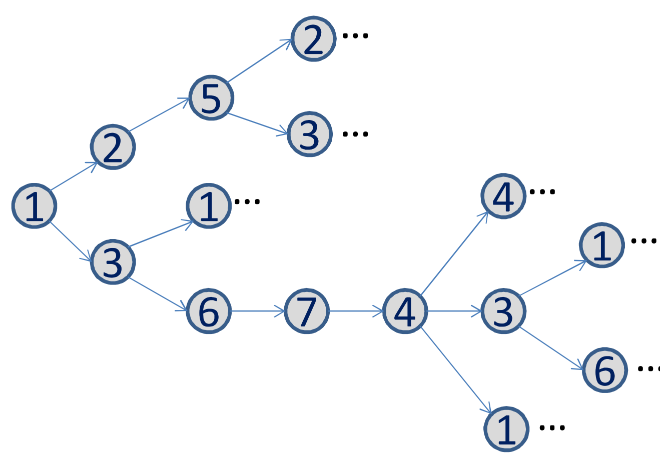
- Finite-state systems modeled by labeled state transition graphs called *Kripke Structures*
  - Pick initial state and unroll to create an infinite computation tree

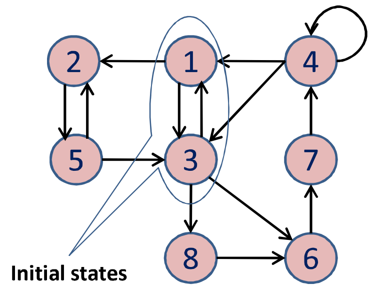


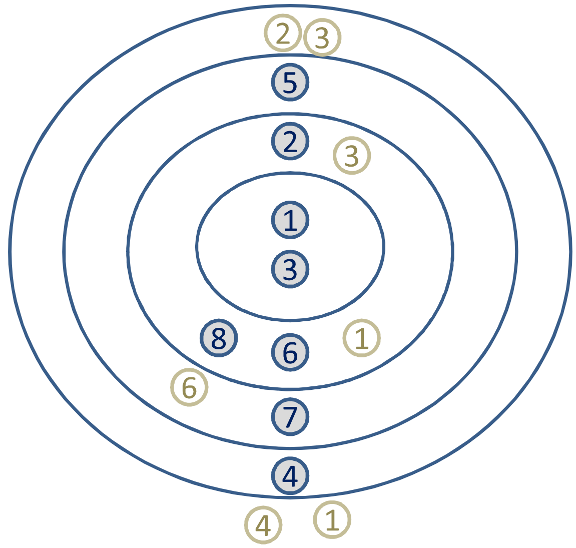
From Ed Clark lecture on temporal logic. Ed is a Turing Award winner (w/Emerson and Sifakis) for his role in developing model checking



h







TLA Model Checking Results

Diameter: Number of states in the longest path in which no state appears twice
States Found: Total number of states examined in a step or a successor state
Distinct States: Number of states in the graph
Queue Size: Number of new states reached that haven't been evaluated

State Graph Construction from Specification

1. Start by setting *G* to the set of all possible initial states.
2. For every state *s* in *G,* compute all possible next states.
   – Substitute values to all unprimed variables in the next-state action.
   – For each new state *t,* add to *G*, if not already present and draw an edge from *s* to *t*
3. Repeat until there are no new edges
4. If process terminates, nodes of *G* consist of all reachable states.

- TLC will used disk space if G and Queue don't fit in memory
  – TLC could run for years before running out of memory

The Die Hard Problem

Obtain exactly 4 gallons of water using:
   a 5 gallon jug, a 3 gallon jug, a water faucet, the ground

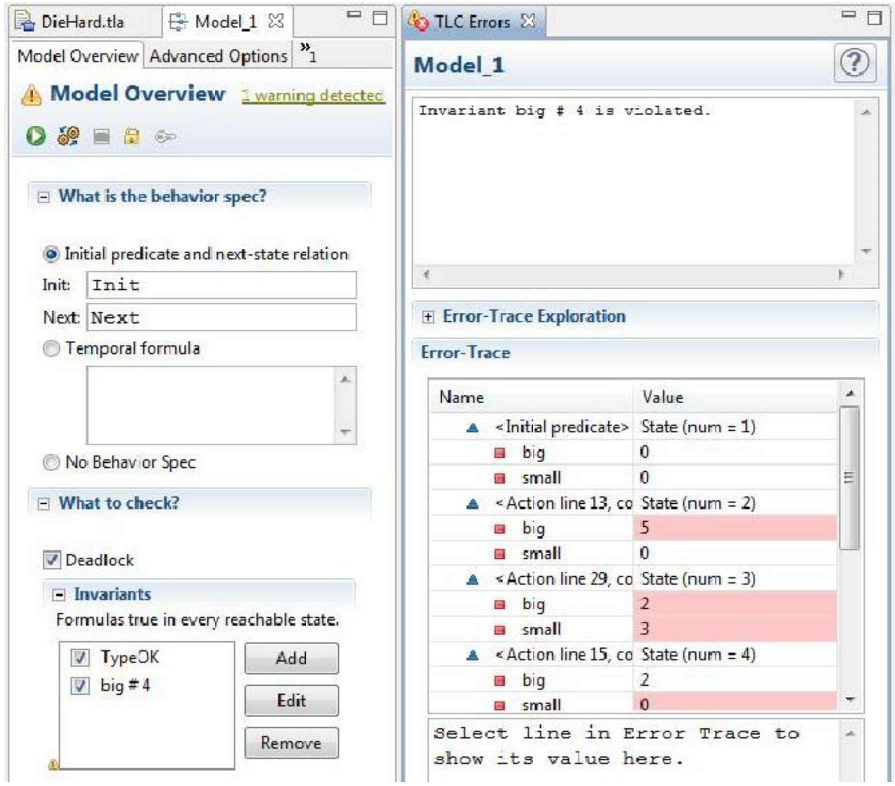
TLA+/TLC modeling
   Variables: the jugs (don't need the faucet or the ground)
   Actions: fill/empty jugs from jugs/faucet to jugs/ground


```
---------------- MODULE DieHard ----------------------
EXTENDS Integers
VARIABLES big, small

Min(m,n) == IF m < n THEN m ELSE n

TypeOK    ==  /\ big   \in 0..5   /\ small \in 0..3
Init      ==  /\ big = 0          /\ small = 0

FillSmall   ==  /\ big' = big   /\ small' = 3
FillBig     ==  /\ big' = 5     /\ small' = small
EmptySmall  ==  /\ big' = big   /\ small' = 0
EmptyBig    ==  /\ big' = 0     /\ small' = small

SmallToBig ==
  LET poured == Min(small, 5-big) IN
    /\ big'    = big + poured
    /\ small'  = small - poured

BigToSmall ==
  LET poured == Min(big, 3-small) IN
    /\ big'    = big - poured
    /\ small'  = small + poured

Next == \/ FillSmall
        \/ FillBig
        \/ EmptySmall
        \/ EmptyBig
        \/ SmallToBig
        \/ BigToSmall
```

Observations from the Die Hard Problem

- Actions are Boolean predicates, not operations
  - Future value of all state variables must be defined for each action
  - Multiple possible future state assignments may be defined
  - In PlusCal, expressions are not actions, but operations
- Easier to read action definitions if:
  - Use let/in statements for intermediate computation values (e.g. *poured*)
  - Don't include future variable values in RHS of comparisons (e.g. *state' = x \/ y' /\z*)
- Applying TLC **Key Lesson**:
  - To obtain a (good or bad) trace add an invariant asserting something doesn't happen!
    - *big#4* added as an invariant to find the sequence of steps the design could take
- Don't re-use, but instead re-write specifications
  - unlike programming where fit new program to existing library

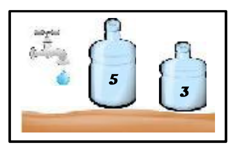
TLA ops



The Die Hard Problem

Obtain exactly 4 gallons of water using:

a 5 gallon jug, a 3 gallon jug, a water faucet, the ground



TLA+/TLC modeling

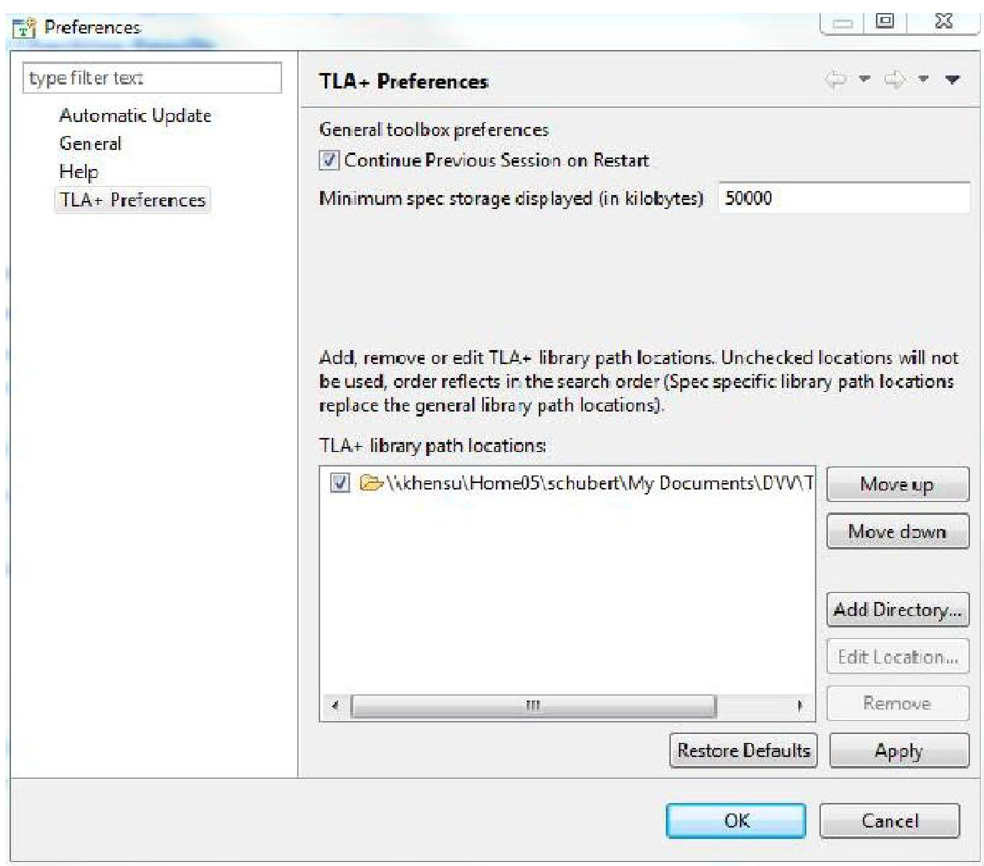
Variables: the jugs (don't need the faucet or the ground)

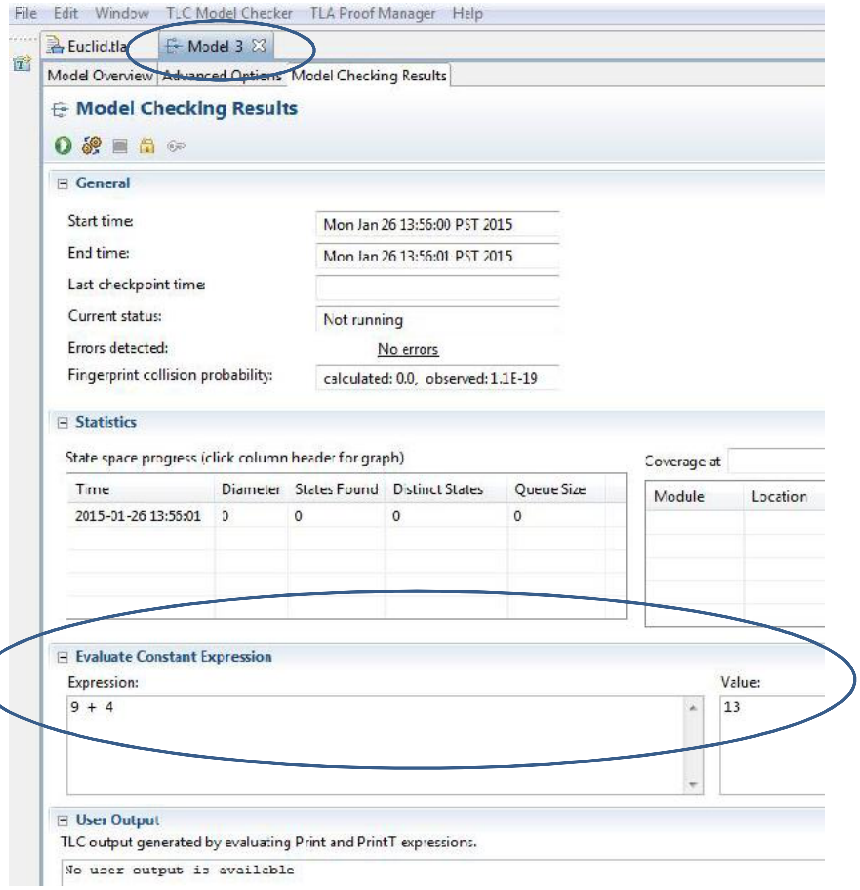
Actions: fill/empty jugs from jugs/faucet to jugs/ground

```
---------------- MODULE DieHard ----------------------
EXTENDS Integers
VARIABLES big, small

Min(m,n) == IF m < n THEN m ELSE n

TypeOK     ==  /\ big  \in 0..5   /\ small \in 0..3
Init       ==  /\ big = 0         /\ small = 0

FillSmall    ==  /\ big' = big    /\ small' = 3
FillBig      ==  /\ big' = 5      /\ small' = small
EmptySmall   ==  /\ big' = big    /\ small' = 0
EmptyBig     ==  /\ big' = 0      /\ small' = small

SmallToBig ==
  LET poured == Min(small, 5-big) IN
    /\ big'    = big + poured
    /\ small'  = small - poured

BigToSmall ==
  LET poured == Min(big, 3-small) IN
    /\ big'    = big - poured
    /\ small'  = small + poured

Next == \/ FillSmall
        \/ FillBig
        \/ EmptySmall
        \/ EmptyBig
        \/ SmallToBig
        \/ BigToSmall
```

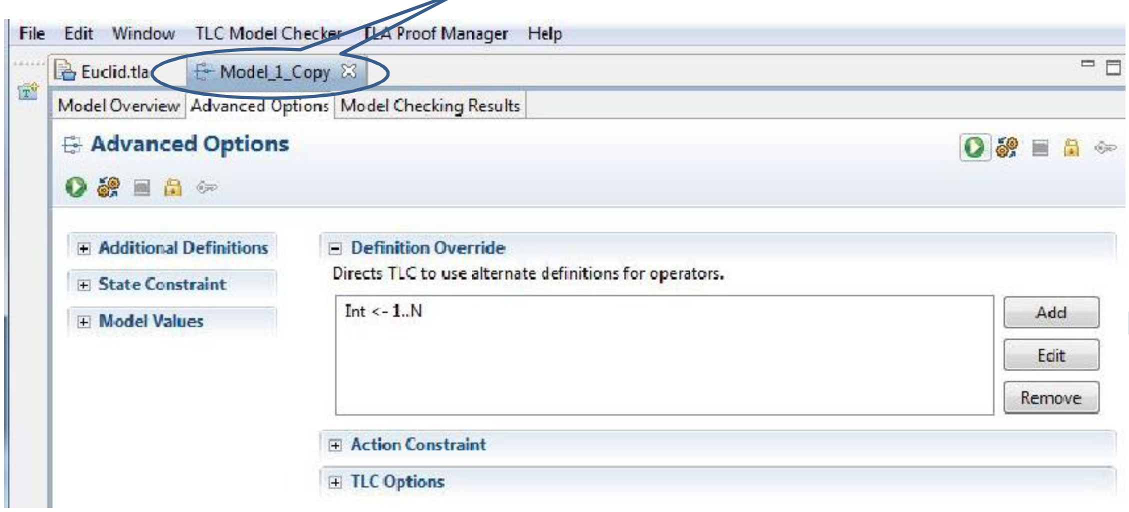
Lessons from Euclid's Algorithm example

- Libraries: TLA+ preferences available to set a path to your libraries
  - Euclid requires predicate definitions for integer divide and GCD
- Using TLA+ as a calculator
  - Create new model (not new spec)
  - Go to *Model Checking Results*, enter expression in *Evaluate Constant Expression* window
- Overriding definitions in TLC
  - Can't check non-enumerable sets --> override definition to make enumerable
  - Even if enumerable, overriding can vastly speed up checking
    - Checking all behaviors of a small model generally more effective at finding errors than checking randomly chosen behaviors
- *CHOOSE* operator CHOOSE x ϵ S : P(x)
  - Find value in S such that P(x) is true, if value exists, else x unspecified
  - Example: CHOOSE i ϵ int : $i^2 = 4$ selects either -2 or 2
- Comments critical, add them!
  - Mathematical specifications are precise, compact, elegant, but hard to comprehend
  - Untyped variable names can describe use of variable, but not its domain





Recommend you override definitions in a cloned model



$$A(p,n) \triangleq \exists q \in Int : n = q*p$$

$$B(n) \triangleq \{p \in Int : A(p, n)\}$$

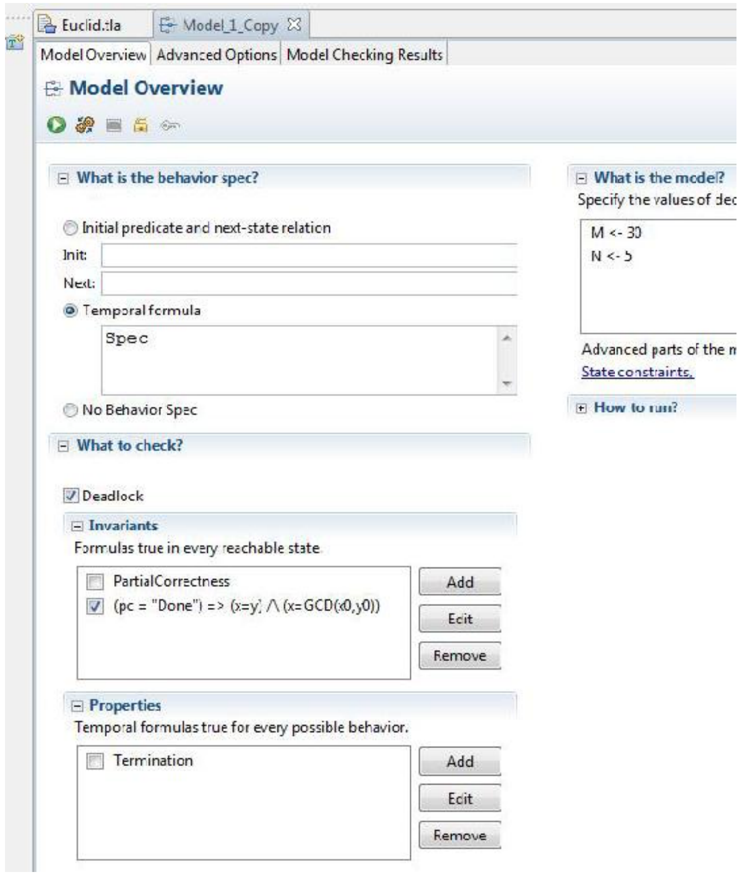
$$C(S) \triangleq CHOOSE\ i \in S : \forall\, j \in S : i \geq j$$

$$D(m,n) \triangleq C(B(m) \cap B(n))$$

Lessons from Euclid's Algorithm example

- Distinction between program (e.g. PlusCal program) and hardware
    - Introduction of pc variable
    - Definition of termination
- Safety and Liveness properties (complementary)
    - Safety property
        - Something bad happens
        - Can be violated in any single step
    - Liveness property
        - Something good happens
        - Not violated in any single step, but by the entire behavior
- Add assertions as invariants or properties in model
- Add assertions to TLA+ specification
    - Operator: Assert(P, m)
        - P is a predicate, m a failure message
    - Requires extending model with module TLC
- Checking Liveness problematic due to stuttering steps

  TLA+ temporal operators: $WF_{vars}(P)$
  $SF_{vars}(P)$



PlusCal translation: grain of atomicity

- PlusCal labels each step
  - Execution is a step from one label to another
  - Translator will add labels as needed
    - Seeks minimum number of steps so as to optimize checking
    - Seeks simplest translation
    - Next-state action evaluates if and body in a single step

```
--fair algorithm Euclid {
 variables x = M, y =N
 { abc: while (x#y)
        {d: if (x<y) {e: y:= y - x}
            else      {x:= x - y}
           };
  assert (x=y) /\ (x=GCD(x0,y0))
 }
}
```

- Rules for where labels go
  - First statement in body of algorithm (required)
  - While statements (required)
  - Any complete statement with label becomes an action
    - This will increase the number of reachable states

Skipping Proofs

- Invariants
- Inductive Invariants
- Proving Euclid *PartialCorrectness* invariant by adding an inductive invariant
  1. *Init => Inv*
  2. *Inv /\ Next => Inv’*
  3. *Inv =>PartialCorrectness*
- Importance of Types:
  – *TypeOK* correctness condition