Checking Concurrent Programs

CS3211 Parallel and Concurrent Programming

Increase the confidence in your concurrent programs

- My code works and I don’t know why
  - No confidence that the code will always work
  - Will making a change break my code?
- How to increase the confidence in your code
  - (Classic testing methods)
  - Sanitizers (in CS3211 lecture 4)
  - Model checking – mathematically proving that the code is correct
    - Many approaches out there
    - Unfortunately, these approaches are not really used in the industry (much)

Model and checking the model

Model checking of a formal specification

- Build the **model** using a special Domain Specific Language (DSL)
  - Write a formal specification, usually done in a new language
- Then **check** the model
  - Are all the constraints met?
  - Does anything unexpected happen?
  - Does it deadlock?
- Why spend time with this?
  - Check things make sense before starting the (costly) implementation
  - Prove certain properties for existing code
  - Allow for aggressive optimizations without compromising correctness

aka “**formal methods**” approach

Formal Methods

Advantages

- Rigorous
- Verify all traces exhaustively
- Produce a system run that violates the requirement

Disadvantages

- The specification used is faulty
- Tedious in coming up with a complete specification
- Time consuming

Approaches to Model Checking

- Write a formal specification of the system and check it
  - Use a model checker or proof assistant
    - Various degrees of automation
    - The checker usually checks the states exhaustively
  - Use the specification to write the code
    - Manually write code
    - Automatically generate code from specification
- Add formal specification (invariants that should hold) in the code, as comments
  - Use a model checker to check the invariants
  - Difficult to make the model checker understand the code
    - Use some symbolic execution
  - Limited in functionalities for concurrent code

Model checkers for concurrent programs

- TLA$^+$ (TLC) - Temporal Logic of Actions+
  - Focusses on temporal properties
  - Good for modeling concurrent systems (and distributed systems)
- Coq Proof Assistant
  - Generates oCaml, Haskell and Scheme
  - Good for interactive proof methods
- Alloy (alloy analyzer)
  - Focusses on relational logic
  - Good for modeling structures

TLA+

- Proposed by Leslie Lamport in 1999
  - Defines TLA+ as a "quixotic attempt to overcome engineers' antipathy towards mathematics"
- High-level language for modeling programs and systems – especially concurrent and distributed ones
- Based on the idea that the best way to describe things precisely is with simple mathematics
- Approach
  - A specification in TLA+ is written
  - The specification is proven (verified) using a checker by exhaustively testing the states
  - Manually write the code based on the TLA+ spec

How it works?

- The model checker finds all possible system behaviours (states) up to some number of execution steps
- Examines the states for violations of desired invariance properties such as **safety** and **liveness**
- TLA+ specifications use basic set theory to define **safety** (bad things won't happen) and temporal logic to define **liveness** (good things eventually happen)

TLA+

- Temporal (time)
- Logic (Boolean logic) of
- Actions (state machines)
- Plus (some stuff)

Boolean Logic

Boolean	Mathematics	TLA+	Programming
AND	$a \land b$	`a /\ b`	`a && b`
OR	$a \lor b$	`a \/ b`	`a || b`
NOT	$\neg a$	`~a`	`!a; not a`

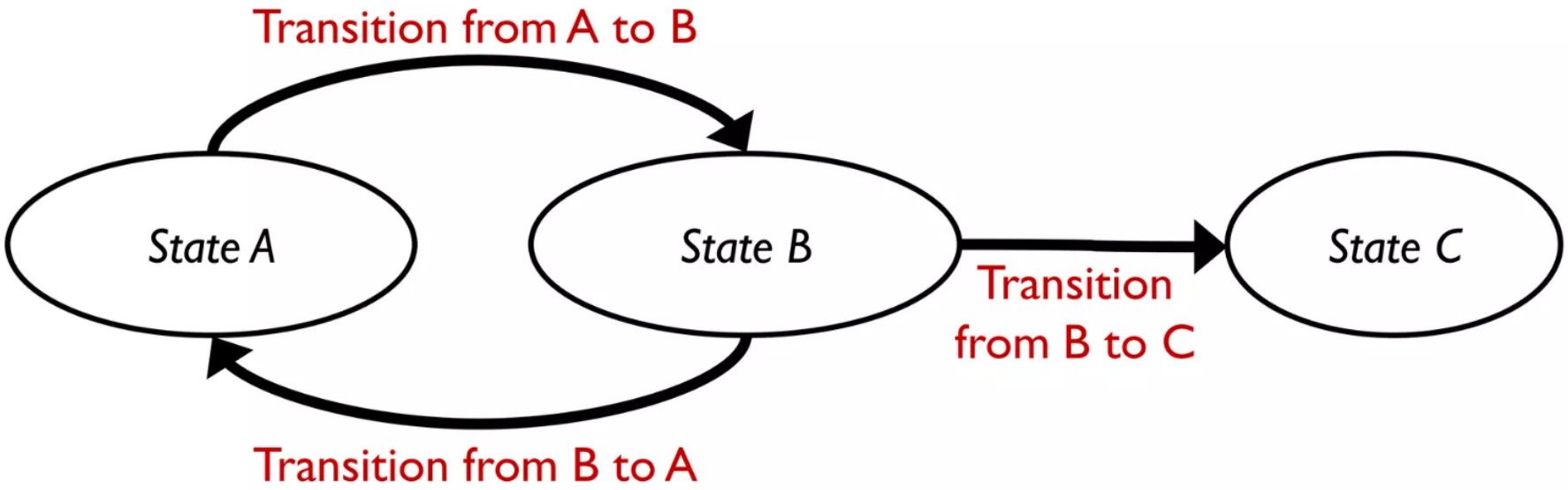
Boolean Logic

- A **predicate** is an expression that returns a Boolean

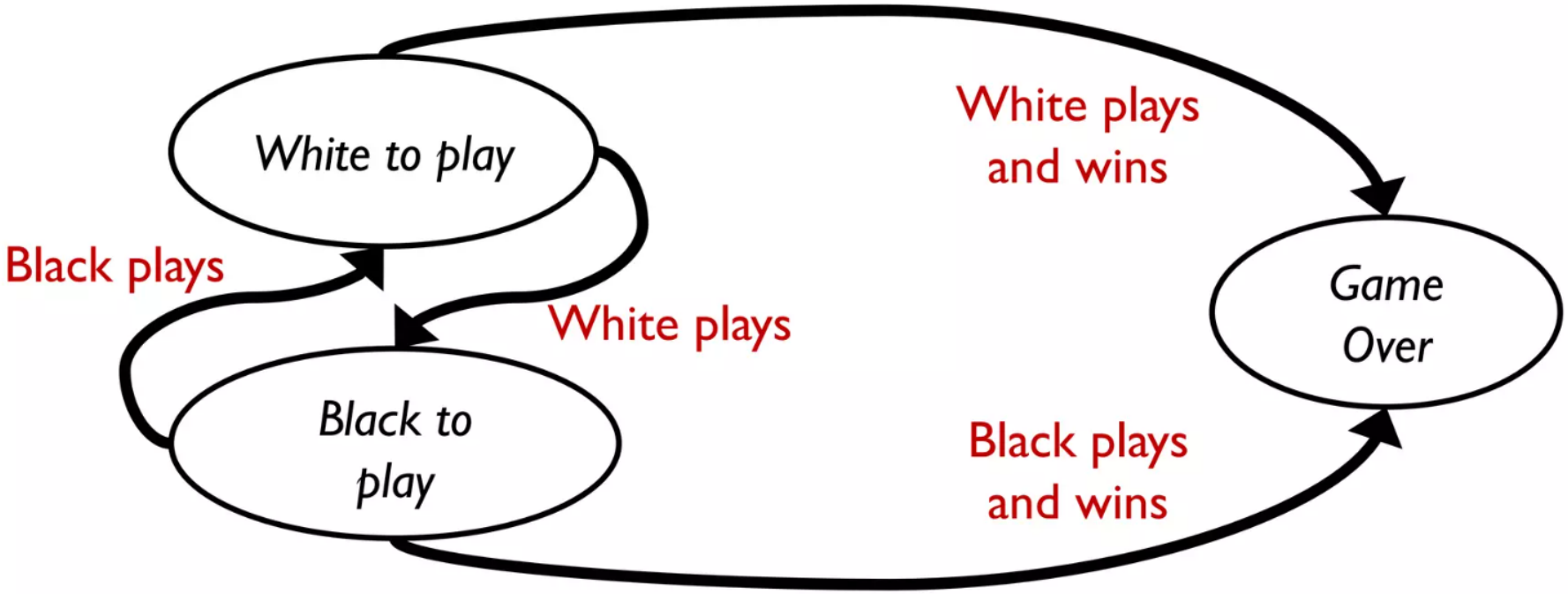
```
// programming language definition
function(a,b,c) {
  (a && b) || (a && !c)
}

\* TLA-style definition
operator(a,b,c) ==
   (a /\ b) \/ (a /\ ~c)
```

Actions – state machines

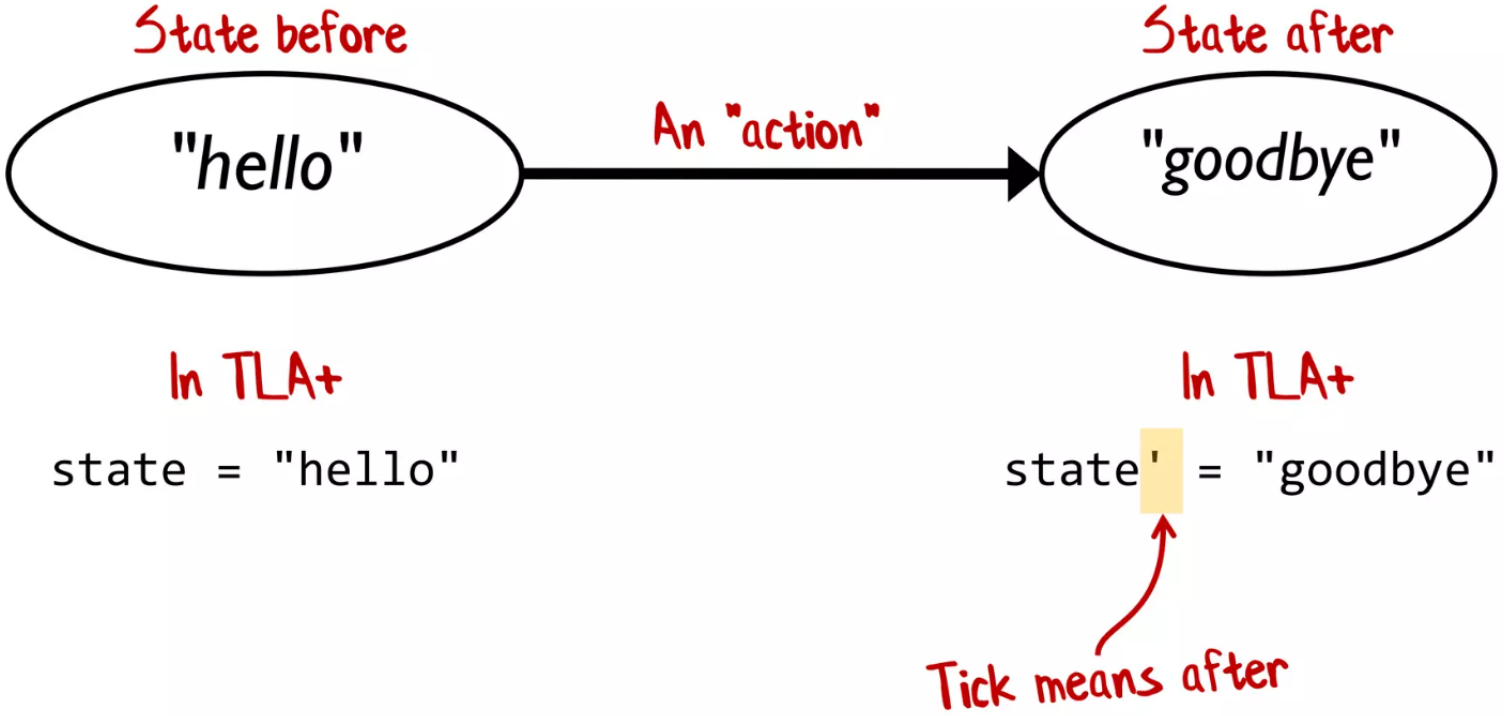
- State machines
  - States
  - Transitions



State machine for playing chess



Formalizing the actions



Formalizing the actions

In TLA+, define the action "Next" like this

```
Next ==
  state = "hello"
  /\ state' = "goodbye"
```

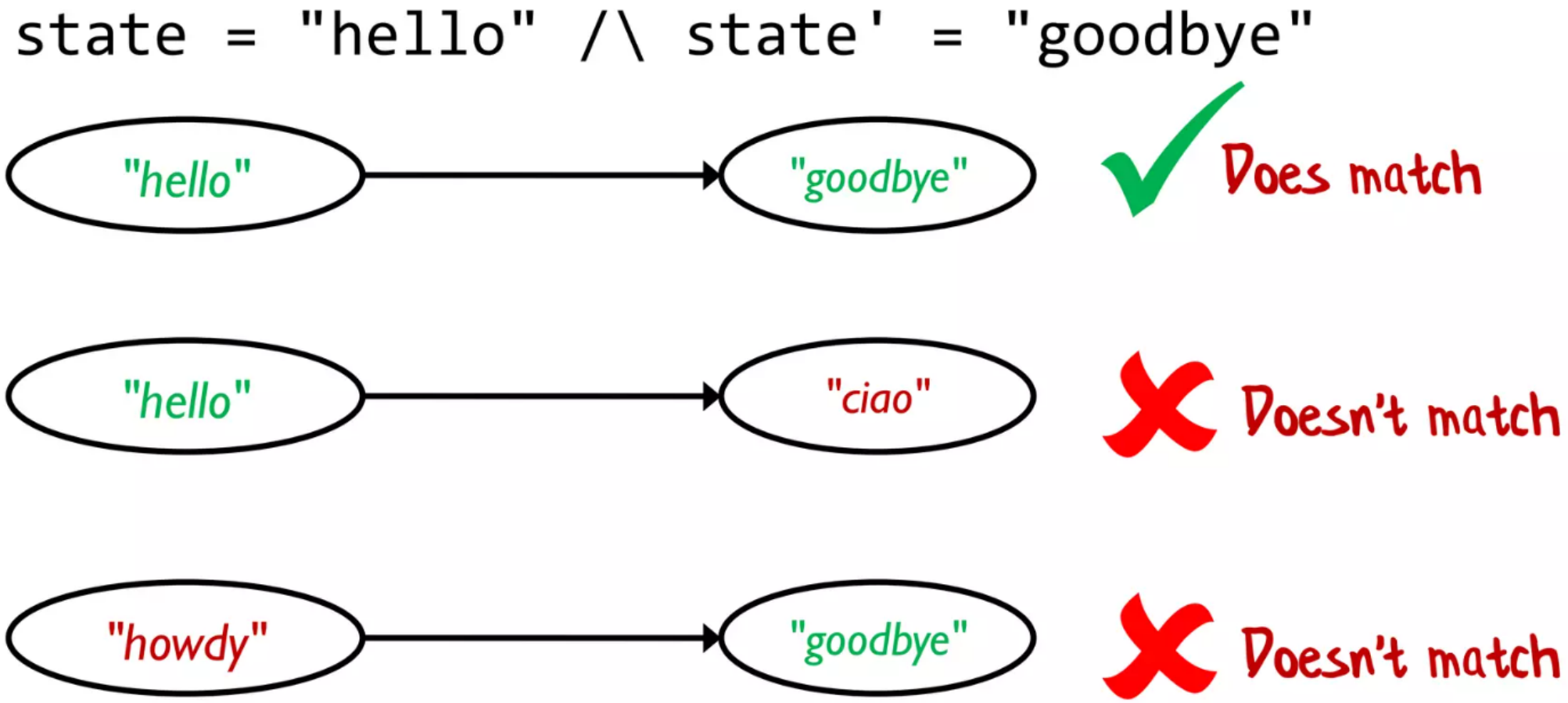
The action is the transition

- This is test, not an assignment!


```
Next ==
  state = "hello"
  /\ state' = "goodbye"
```

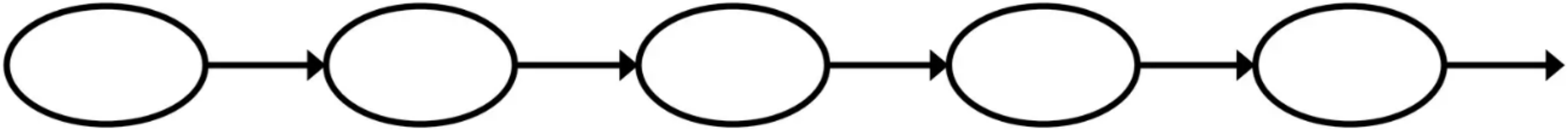
This is not an assignment!

Actions are tests



Temporal – state transitions over time

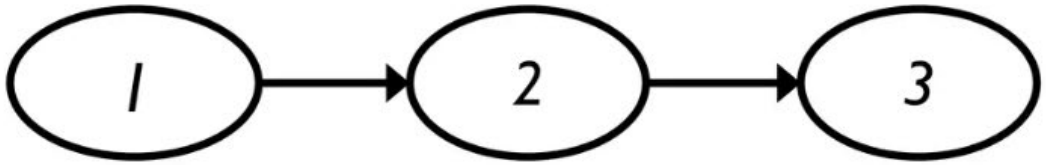
- Infinite amount of time



- TLA+ can ask questions like:
  - Is something always true?
  - Is something ever true?
  - If X happens, must Y happen afterwards?

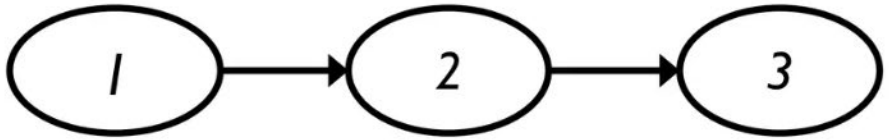
  over time

Count to three



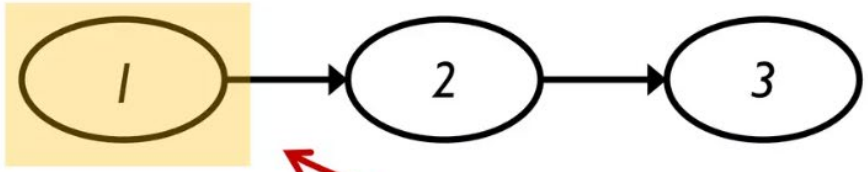

```
// programming language version
var x = 1
x = 2
x = 3
```

Count to three




```
\* TLA version
Init ==                    \* initial state
  x=1

Next ==                    \* transition
  (x=1 /\ x'=2)            \* match step 1
  \/ (x=2 /\ x'=3)         \* or match step 2
```

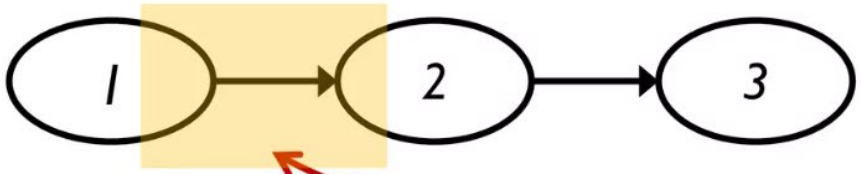
Count to three



Initial state

```
Init ==
  x=1

Next ==
  (x=1 /\ x'=2)      \* match step 1
  \/ (x=2 /\ x'=3)   \* or match step 2
```

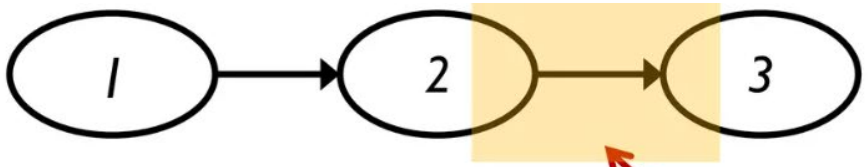
Count to three



Matches first transition

```
Init ==
  x=1

Next ==
  (x=1 /\ x'=2)      \* match step 1
  \/ (x=2 /\ x'=3)   \* or match step 2
```

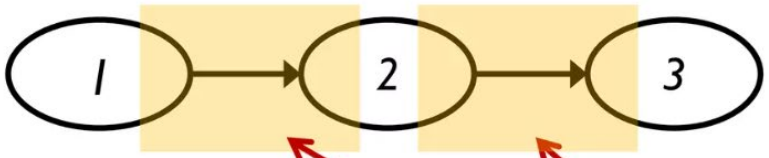
Count to three



Matches second transition

```
Init ==
  x=1

Next ==
  (x=1 /\ x'=2)      \* match step 1
  \/ (x=2 /\ x'=3)   \* or match step 2
```

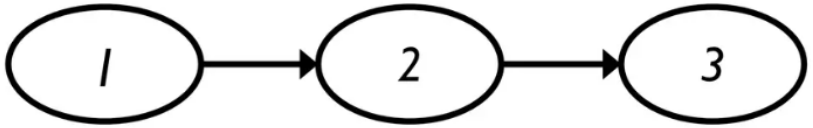
Count to three



Matches first OR second transition

```
Init ==
  x=1

Next ==
  (x=1 /\ x'=2)      \* match step 1
  \/ (x=2 /\ x'=3)   \* or match step 2
```

Count to three, refactored

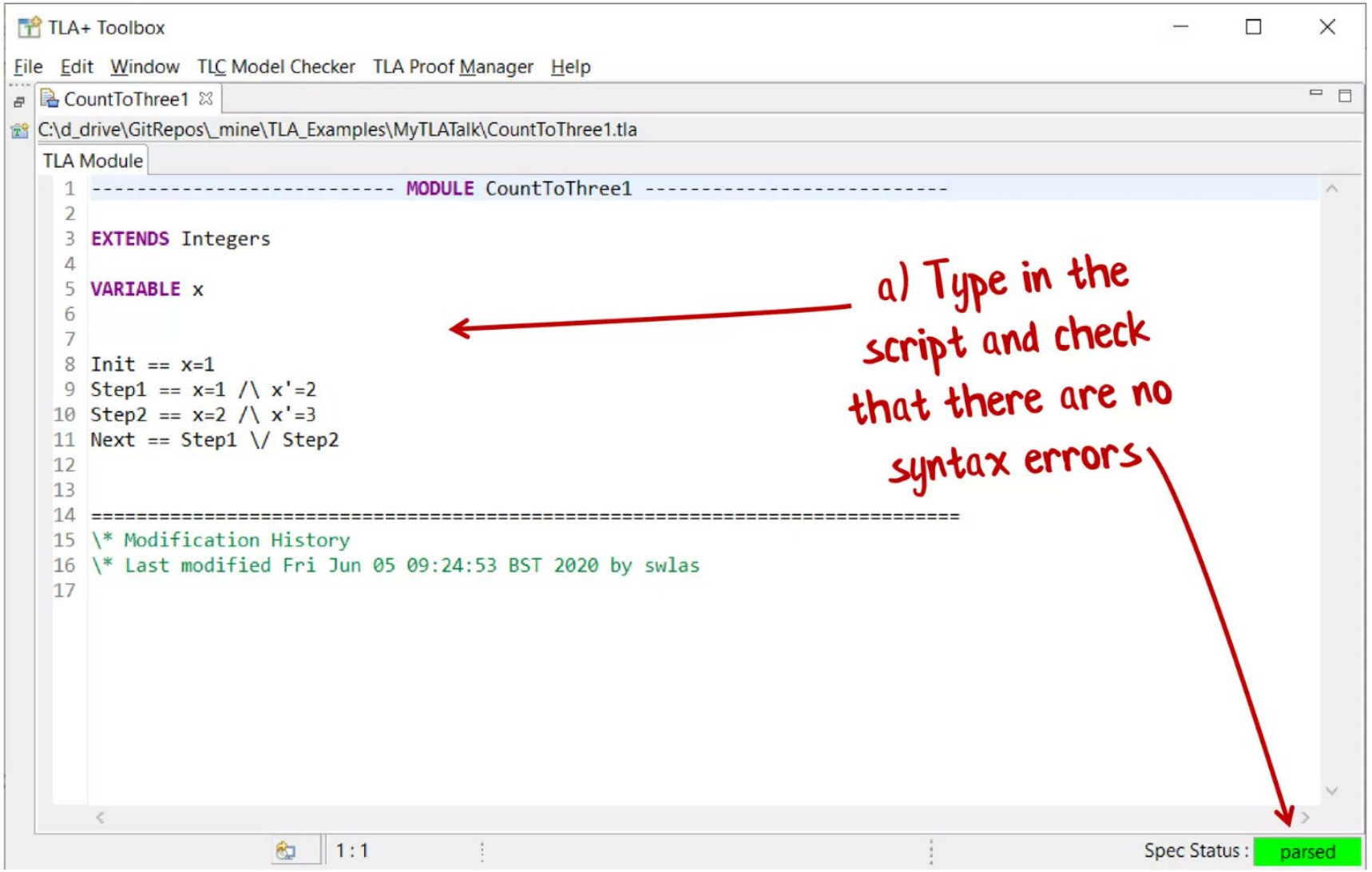


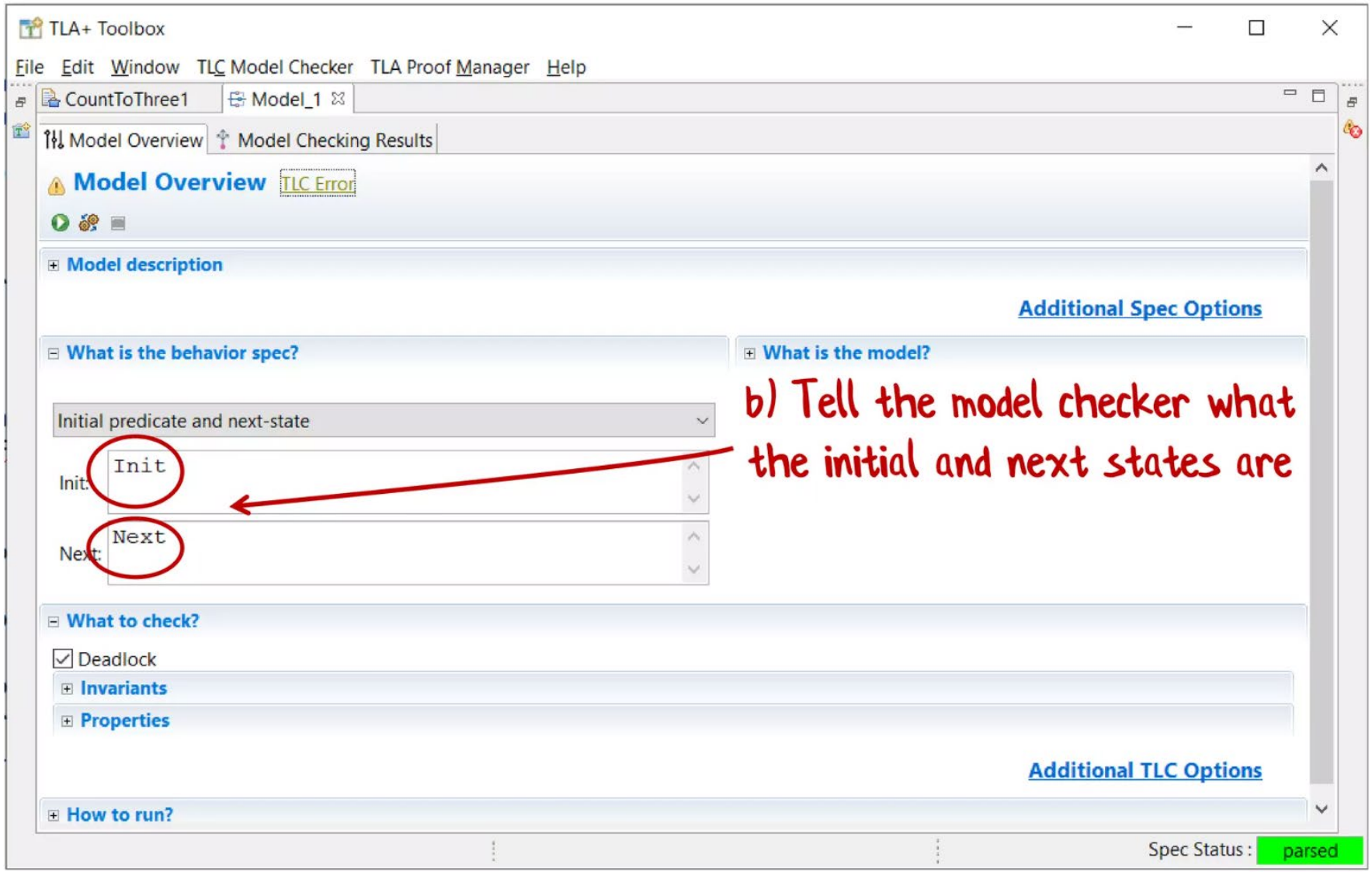

```
Init == x=1
Step1 == x=1 /\ x'=2
Step2 == x=2 /\ x'=3
Next == Step1 \/ Step2
```

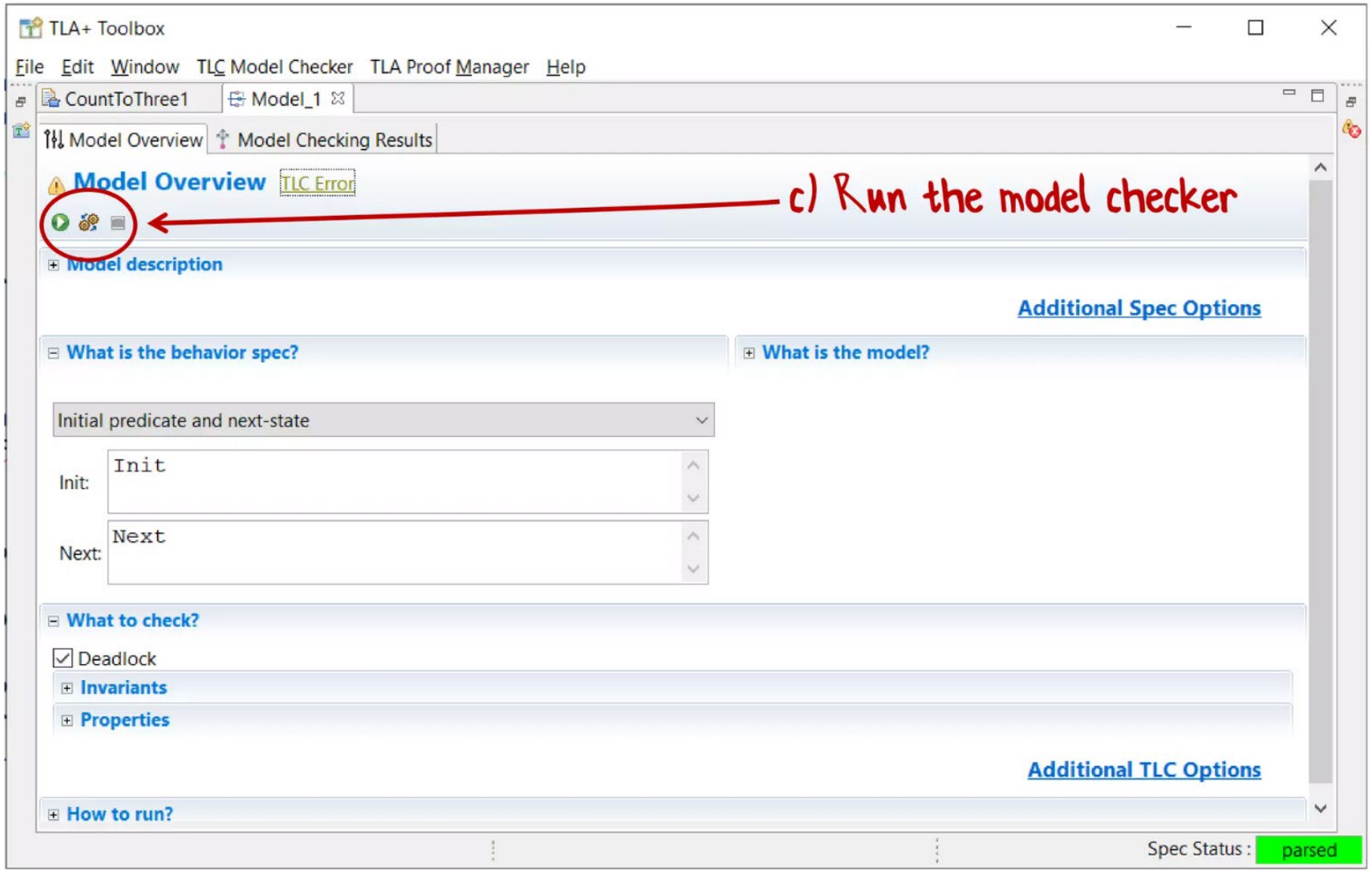
These are called "operators".
They are like macros,
not procedures/functions

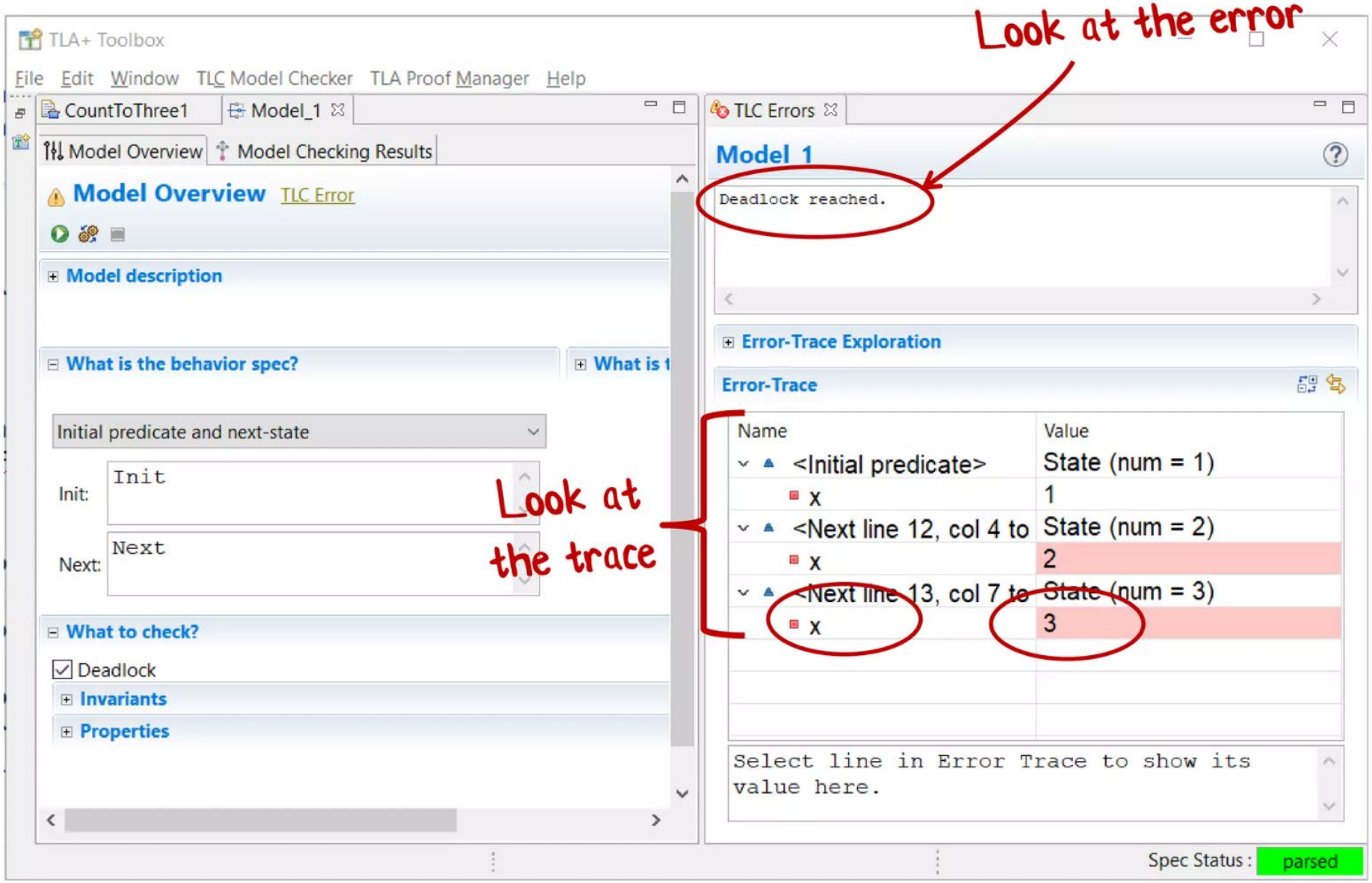
Refactored version.
Steps are now explicitly named

TLA+ toolbox: IDE and checker

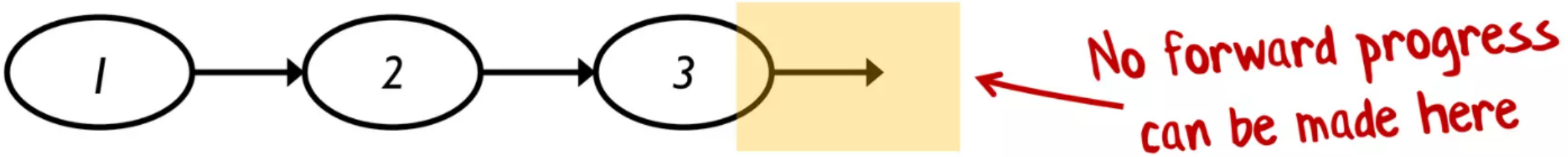



TLA+ Toolbox
File Edit Window TLC Model Checker TLA Proof Manager Help
CountToThree1
Model_1
Model Overview
Model Checking Results
Model Overview
TLC Error
Model description
Additional Spec Options
What is the behavior spec?
What is the model?
Initial predicate and next-state
Init:
Init
Next:
Next
b) Tell the model checker what the initial and next states are
What to check?
Deadlock
Invariants
Properties
Additional TLC Options
How to run?
Spec Status :
parsed

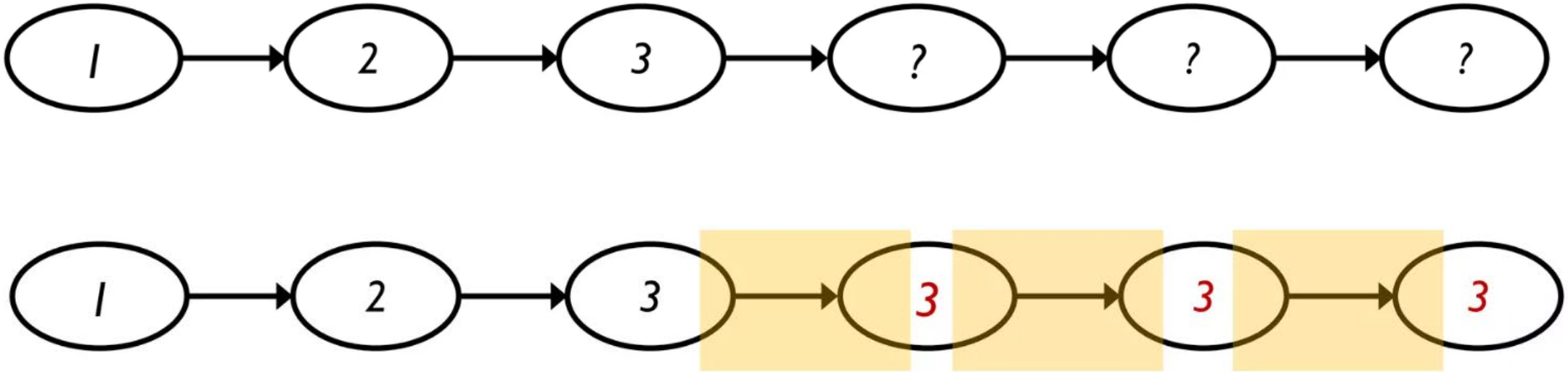

TLA+ Toolbox
File Edit Window TLC Model Checker TLA Proof Manager Help
CountToThree1
Model_1
Model Overview
Model Checking Results
Model Overview
TLC Error
c) Run the model checker
Model description
Additional Spec Options
What is the behavior spec?
What is the model?
Initial predicate and next-state
Init:
Init
Next:
Next
What to check?
Deadlock
Invariants
Properties
Additional TLC Options
How to run?
Spec Status :
parsed


Look at the error
TLA+ Toolbox
File Edit Window TLC Model Checker TLA Proof Manager Help
CountToThree1
Model_1
Model Overview
Model Checking Results
Model Overview
TLC Error
Model description
What is the behavior spec?
What is t
Initial predicate and next-state
Init:
Init
Next:
Next
Look at the trace
What to check?
Deadlock
Invariants
Properties
TLC Errors
Model 1
Deadlock reached.
Error-Trace Exploration
Error-Trace
Name
Value
<Initial predicate>
State (num = 1)
x
1
<Next line 12, col 4 to
State (num = 2)
x
2
<Next line 13, col 7 to
State (num = 3)
x
3
Select line in Error Trace to show its value here.
Spec Status :
parsed

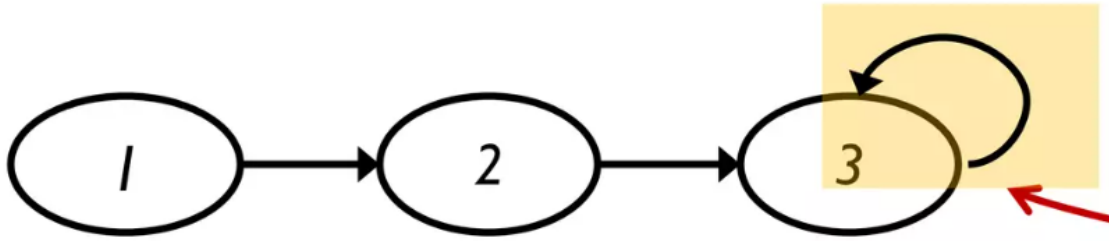
Deadlock in TLA+



- Infinite time in TLA+



Count to three, updated



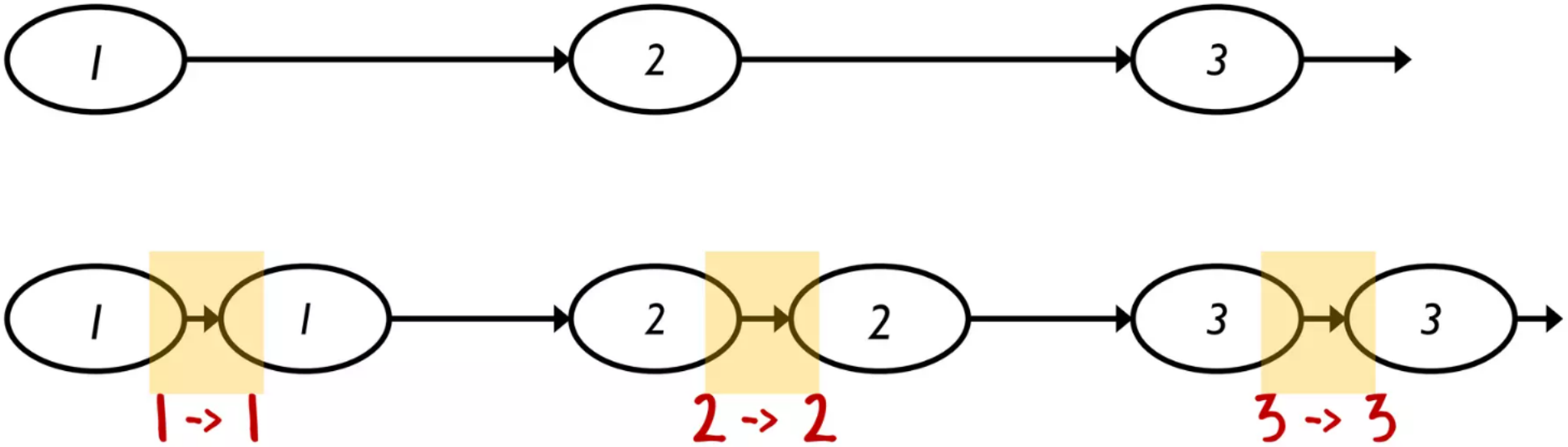
Loop forever here

```
Init == x=1
Step1 == x=1 /\ x'=2
Step2 == x=2 /\ x'=3
Done == x=3 /\ UNCHANGED x
Next == Step1 \/ Step2 \/ Done
```

If we run this now, we no longer get a deadlock

Add a final step to the model

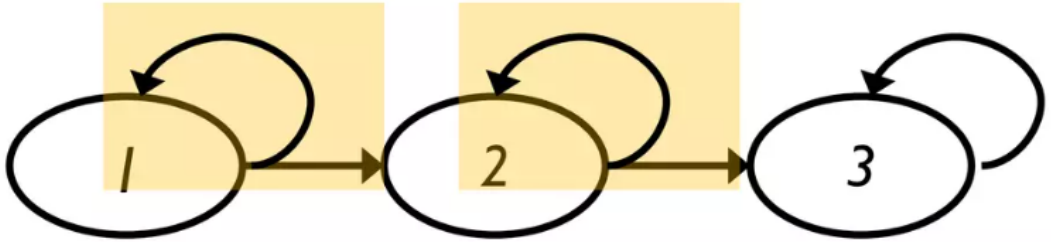
Doing nothing is always an option!



Staying in the same state is
almost always a valid state transition!

The TLA+ term for this is "stuttering"

Count to three, with stuttering




```
Init == x=1
Step1 == x=1 /\ x'=2
Step2 == x=2 /\ x'=3
Done == x=3 /\ UNCHANGED x
Next == Step1 \/ Step2 \/ Done \/ UNCHANGED x
```

With added stuttering

The power of temporal properties

- A **property** applies to the whole system over time
  - Not just to individual states
- Checking these properties is important
  - Humans are bad at this
  - Programming languages are bad at this too
  - TLA+ can help with this!

Properties in TLA+

- Always true
  - For all tests, x>0
- Eventually true
  - At some point in time, x=2
- Eventually always
  - Eventually becomes true (done) and stays there (done)
  - x eventually becomes 3 and then stays there
- Leads to
  - If x ever becomes 2, then it will become 3 later

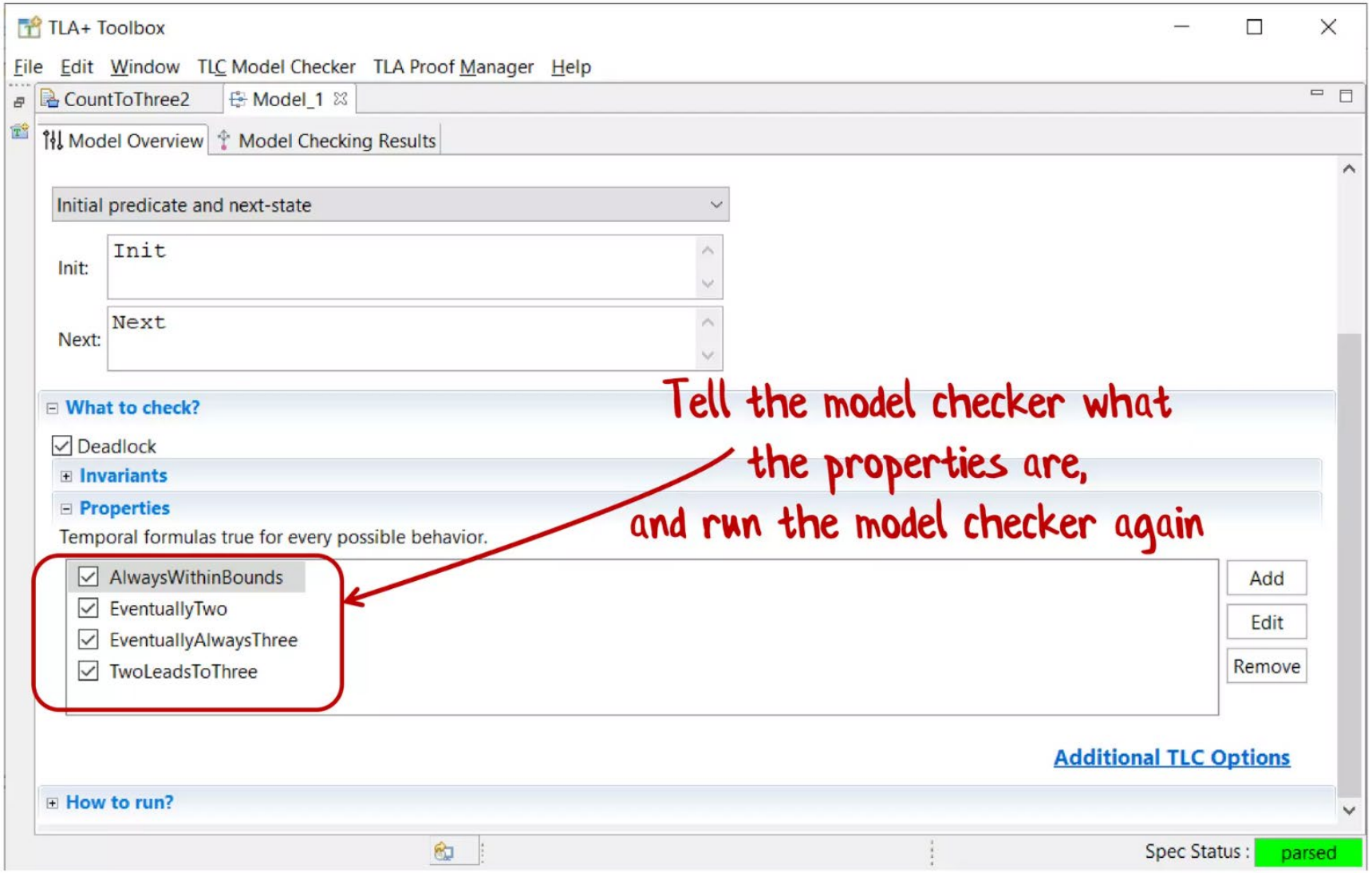
Properties for “count to three”

In English	Formally	In TLA+
x is always > 2	Always (x > 0)	`[] (x > 0)`
At some point x is 2	Eventually (x = 2)	`<> (x = 2)`
x eventually becomes 3 and then stays there.	Eventually (Always (x = 3))	`<>[] (x = 3)`
if x ever becomes 2 then it will become 3 later.	(x=2) leads to (x=3)	`(x=2) ~> (x=3)`

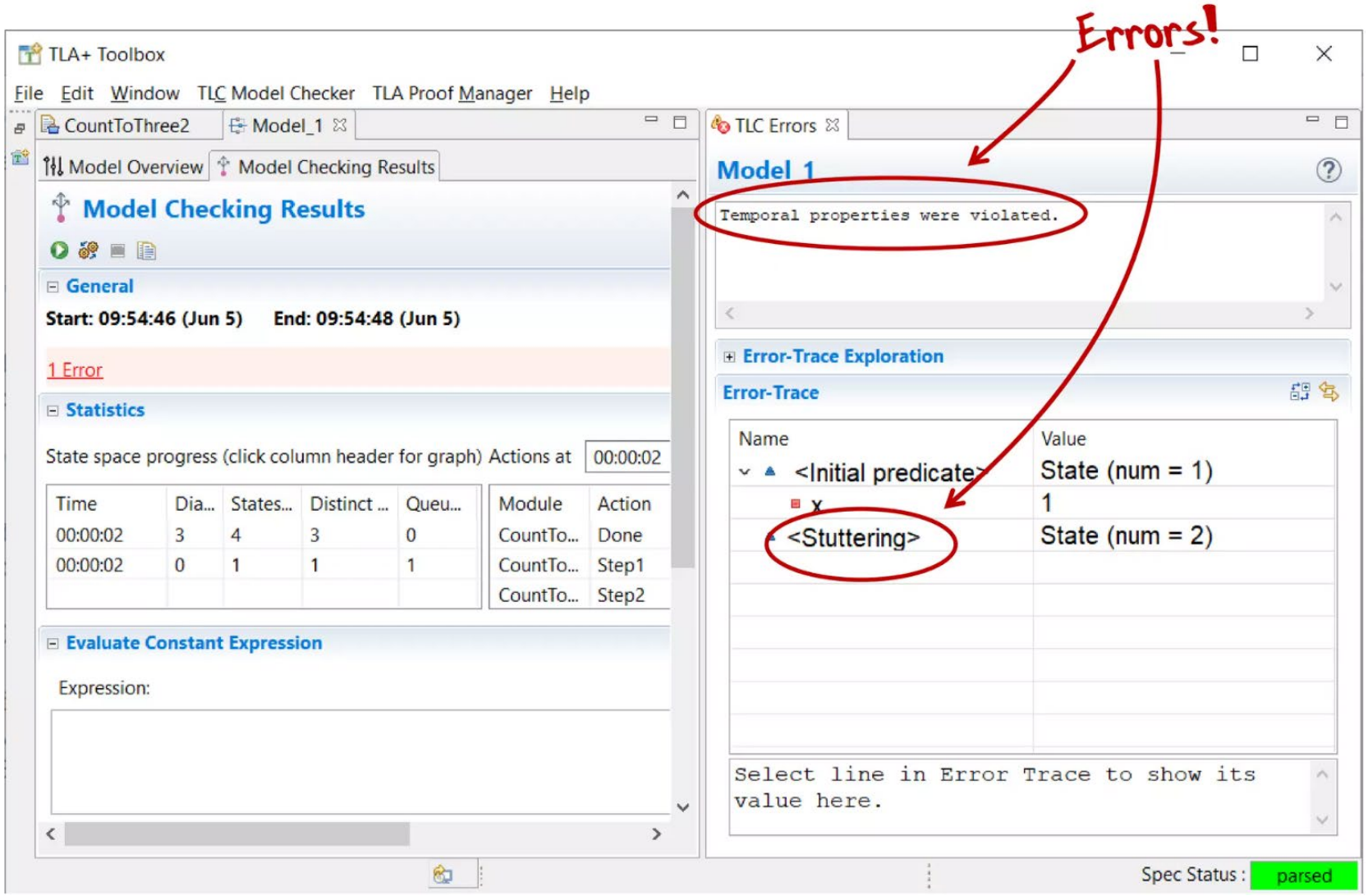
Adding properties to the script

```
\* Always, x >= 1 && x <= 3
AlwaysWithinBounds == [](x >= 1 /\ x <= 3)

\* At some point, x = 2
EventuallyTwo == <>(x = 2)

\* At some point, x = 3 and stays there
EventuallyAlwaysThree == <>[](x = 3)

\* Whenever x=2, then x=3 later
TwoLeadsToThree == (x = 2) ~> (x = 3)
```

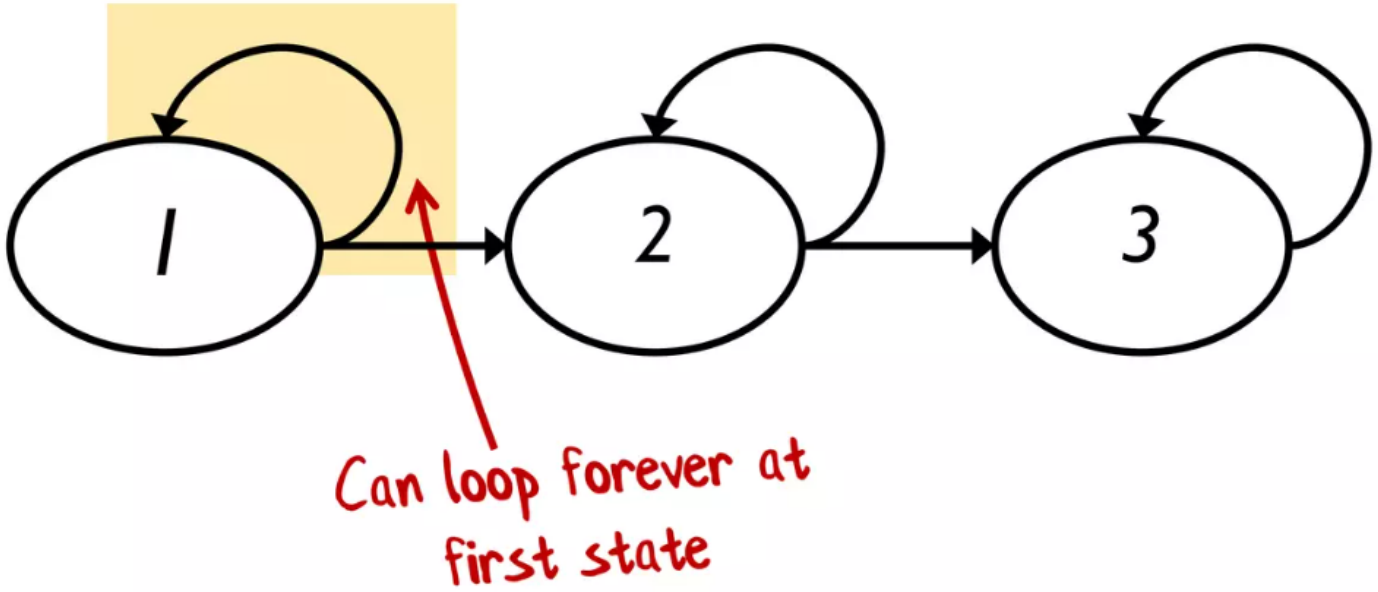
Adding properties to the script



Oh no! Model checker says we have errors!



Stuttering caused a loop!



So only the "AlwaysWithinBounds" property is true

Fixing the error

- Make sure every possible transition is followed
- Don’t get stuck in an infinite loop

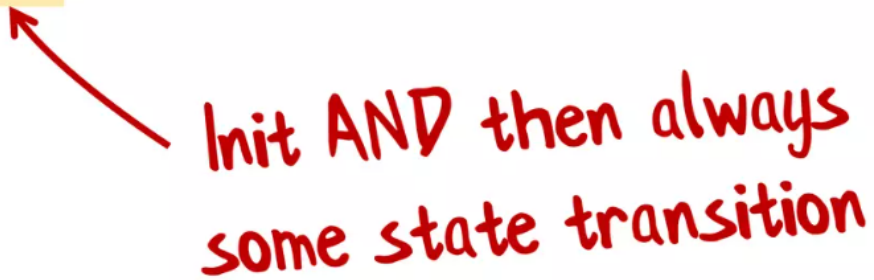
Add fairness! TLA+ can model this

Refactor #1: change the spec to merge init/next

```
Init == x=1
Step1 == x=1 /\ x'=2
Step2 == x=2 /\ x'=3
Done == x=3 /\ UNCHANGED x
Next == Step1 \/ Step2 \/ Done

Spec = Init /\ [](Next \/ UNCHANGED x)
```

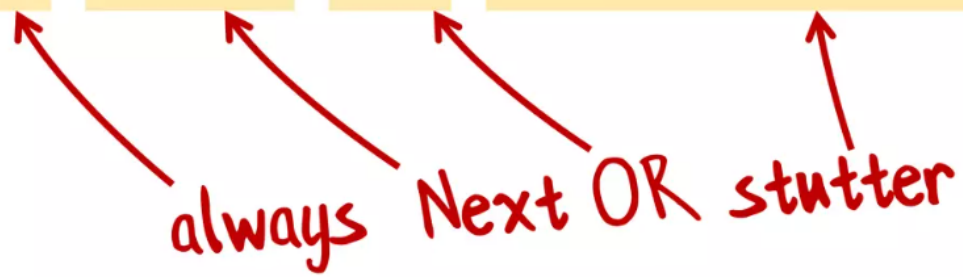
Refactor #1: change the spec to merge init/next

```
Init == x=1
Step1 == x=1 /\ x'=2
Step2 == x=2 /\ x'=3
Done == x=3 /\ UNCHANGED x
Next == Step1 \/ Step2 \/ Done

Spec = Init /\ [](Next \/ UNCHANGED x)
```



Refactor #1: change the spec to merge init/next

```
Init == x=1
Step1 == x=1 /\ x'=2
Step2 == x=2 /\ x'=3
Done == x=3 /\ UNCHANGED x
Next == Step1 \/ Step2 \/ Done

Spec = Init /\ [](Next \/ UNCHANGED x)
```



Refactor #2: Use a special syntax for stuttering

Before

```
Spec = Init /\ [](Next \/ UNCHANGED x)
```

After

```
Spec = Init /\ [][Next]_x
```

Special syntax for "Next or stutter on x"

Refactor #3: Now we can add fairness!

```
Spec = Init /\ [][Next]_x
```

With fairness

```
Spec = Init /\ [][Next]_x /\ WF_x(Next)
```

If "Next" step is available, eventually try it, ignoring stutter

The new "weak" fairness condition

The complete spec with fairness

```
Init == x=1
Step1 == x=1 /\ x'=2
Step2 == x=2 /\ x'=3
Done == x=3 /\ UNCHANGED x
Next == Step1 \/ Step2 \/ Done
Spec == Init /\ [][Next]_x /\ WF_x(Next)

\* properties to check
AlwaysWithinBounds == [](x >= 1 /\ x <= 3)
EventuallyTwo == <>(x = 2)
EventuallyAlwaysThree == <>[](x = 3)
TwoLeadsToThree == (x = 2) ~> (x = 3)
```

If we run this new spec, all four temporal properties are now true!

The complete spec with fairness

```
Init == x=1
Step1 == x=1 /\ x'=2
Step2 == x=2 /\ x'=3
Done == x=3 /\ UNCHANGED x
Next == Step1 \/ Step2 \/ Done
Spec == Init /\ [][Next]_x /\ WF_x(Next)

\* properties to check
AlwaysWithinBounds == [](x >= 1 /\ x <= 3)
EventuallyTwo == <>(x = 2)
EventuallyAlwaysThree == <>[](x = 3)
TwoLeadsToThree == (x = 2) ~> (x = 3)
```

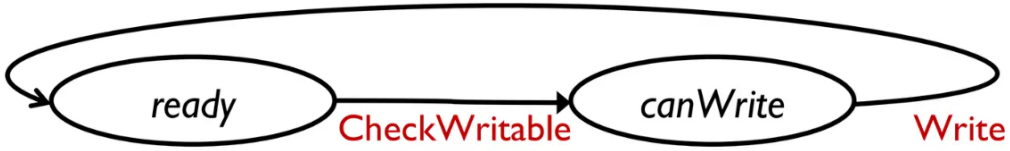
A more complicated example

- Very exciting: we can count to three!
- What about a more complicated problem?
- What about concurrency?
  - Property checking is where TLA+ is powerful and it can help

Producer/consumer problem

Producer:

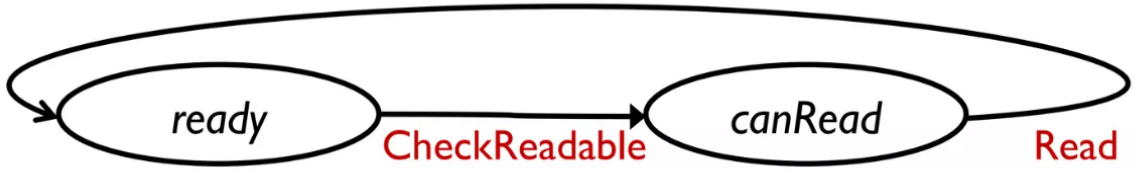
- Check if queue is not full
- If true, then write item to queue



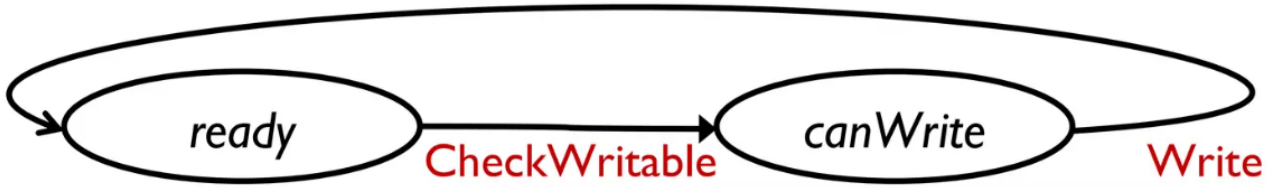
We're choosing to model this as two distinct state transitions, not one atomic step

Consumer:

- Check if queue is not empty
- If true, then read item from queue



States for a Producer

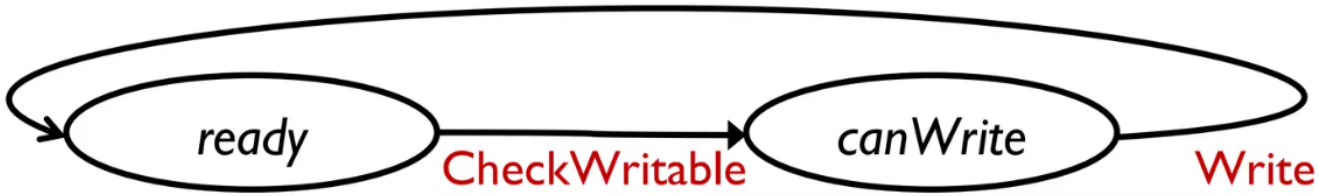



```
def CheckWritable():
  if (queueSize < MaxQueueSize)
     && (producerState = "ready")
  then
     producerState = "canWrite";

def Write():
  if producerState = "canWrite"
  then
     producerState = "ready";
     queueSize = queueSize + 1;
```

Programming language version

Don't care about the contents of the queue

States for a Producer



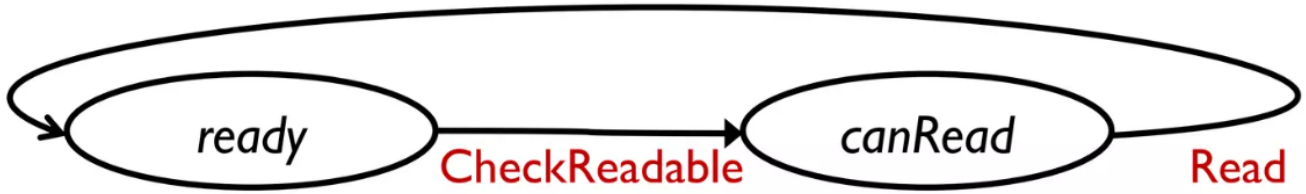
TLA+ version

```
CheckWritable ==
  producerState = "ready"
  /\ queueSize < MaxQueueSize
  /\ producerState' = "canWrite" \* transition
  /\ UNCHANGED queueSize

Write ==
  producerState = "canWrite"
  /\ producerState' = "ready"    \* transition
  /\ queueSize' = queueSize + 1  \* push to queue

ProducerAction == CheckWritable \/ Write
```

Important: All variables must be accounted for in an action

All the valid actions for a producer

States for a Consumer




```
CheckReadable ==
  consumerState = "ready"
  /\ queueSize > 0
  /\ consumerState' = "canRead"  \* transition
  /\ UNCHANGED queueSize

Read ==
  consumerState = "canRead"
  /\ consumerState' = "ready"    \* transition
  /\ queueSize' = queueSize - 1  \* pop from queue

ConsumerAction == CheckReadable \/ Read
```

All the valid actions for a consumer

Complete TLA+ script (1/2)

```
VARIABLES
  queueSize,
  producerState,
  consumerState

MaxQueueSize == 2 \* can be small

Init ==
  queueSize = 0
  /\ producerState = "ready"
  /\ consumerState = "ready"
```

```
CheckWritable ==
  producerState = "ready"
  /\ queueSize < MaxQueueSize
  /\ producerState' = "canWrite"
  /\ UNCHANGED queueSize
  /\ UNCHANGED consumerState

Write ==
  producerState = "canWrite"
  /\ producerState' = "ready"
  /\ queueSize' = queueSize + 1
  /\ UNCHANGED consumerState

ProducerAction ==
  CheckWritable \/ Write
```

Complete TLA+ script (2/2)

```
CheckReadable ==
  consumerState = "ready"
  /\ queueSize > 0
  /\ consumerState' = "canRead"
  /\ UNCHANGED queueSize
  /\ UNCHANGED producerState

Read ==
  consumerState = "canRead"
  /\ consumerState' = "ready"
  /\ queueSize' = queueSize - 1
  /\ UNCHANGED producerState

ConsumerAction ==
  CheckReadable \/ Read
```

```
Next ==
  ProducerAction
  \/ ConsumerAction
```

Non-determinism for free!
No need to create threads, actors, etc

Embedded concurrency!

Complete TLA+ script (2/2)

```
CheckReadable ==
  consumerState = "ready"
  /\ queueSize > 0
  /\ consumerState' = "canRead"
  /\ UNCHANGED queueSize
  /\ UNCHANGED producerState

Read ==
  consumerState = "canRead"
  /\ consumerState' = "ready"
  /\ queueSize' = queueSize - 1
  /\ UNCHANGED producerState

ConsumerAction ==
  CheckReadable \/ Read
```

```
Next ==
  ProducerAction
  \/ ConsumerAction
```

Complete TLA+ script (2/2)

```
CheckReadable ==
  consumerState = "ready"
  /\ queueSize > 0
  /\ consumerState' = "canRead"
  /\ UNCHANGED queueSize
  /\ UNCHANGED producerState

Read ==
  consumerState = "canRead"
  /\ consumerState' = "ready"
  /\ queueSize' = queueSize - 1
  /\ UNCHANGED producerState

ConsumerAction ==
  CheckReadable \/ Read
```

```
Next ==
  ProducerAction
  \/ ConsumerAction
  \/ (UNCHANGED producerState
      /\ UNCHANGED consumerState
      /\ UNCHANGED queueSize)
```

Doing nothing could represent producer or consumer crashing!

Temporal properties for the producer/consumer

```
AlwaysWithinBounds ==
    [] (queueSize >= 0
        /\ queueSize <= MaxQueueSize)
```

- 8 states, no errors
- BUT only for 1 producer and 1 consumer!

Concurrent version with multiple producers/consumers

- Use the **Plus** in TLA+

- We need
  - A **set** of producers
  - A **set** of consumers
- Use the set-description part of TLA+

```
producers={"p1","p2"}
consumers={"c1","c2"}
```

Plus… set theory!

Set theory	Mathematics	TLA+	Programming
e is an element of set S	e ∈ S	`e \in S`	
Define a set by enumeration	{1,2,3}	`{1,2,3}`	`[1,2,3]`
Define a set by predicate "p"	{ e ∈ S | p }	`{e \in S : p}`	`Set.filter(p)`
For all e in Set, some predicate "p" is true	∀ e ∈ S : p	`\A e \in S : p`	`Set.all(p)`
There exists e in Set such that some predicate "p" is true	∃ e ∈ S : p	`\E x \in S : p`	`Set.any(p)`

Producer/Consumer Spec, part 1

```
CONSTANT producers, consumers
\* e.g
\*   2 producers={"p1","p2"}
\*   2 consumers={"c1","c2"}

VARIABLES queueSize, producerState, consumerState

MaxQueueSize == 2

Init ==
  queueSize = 0
  /\ producerState = [p \in producers |-> "ready"]
              \* same as {"p1":"ready","p2":"ready"}
  /\ consumerState = [c \in consumers |-> "ready"]
```

Producer/Consumer Spec, part 1

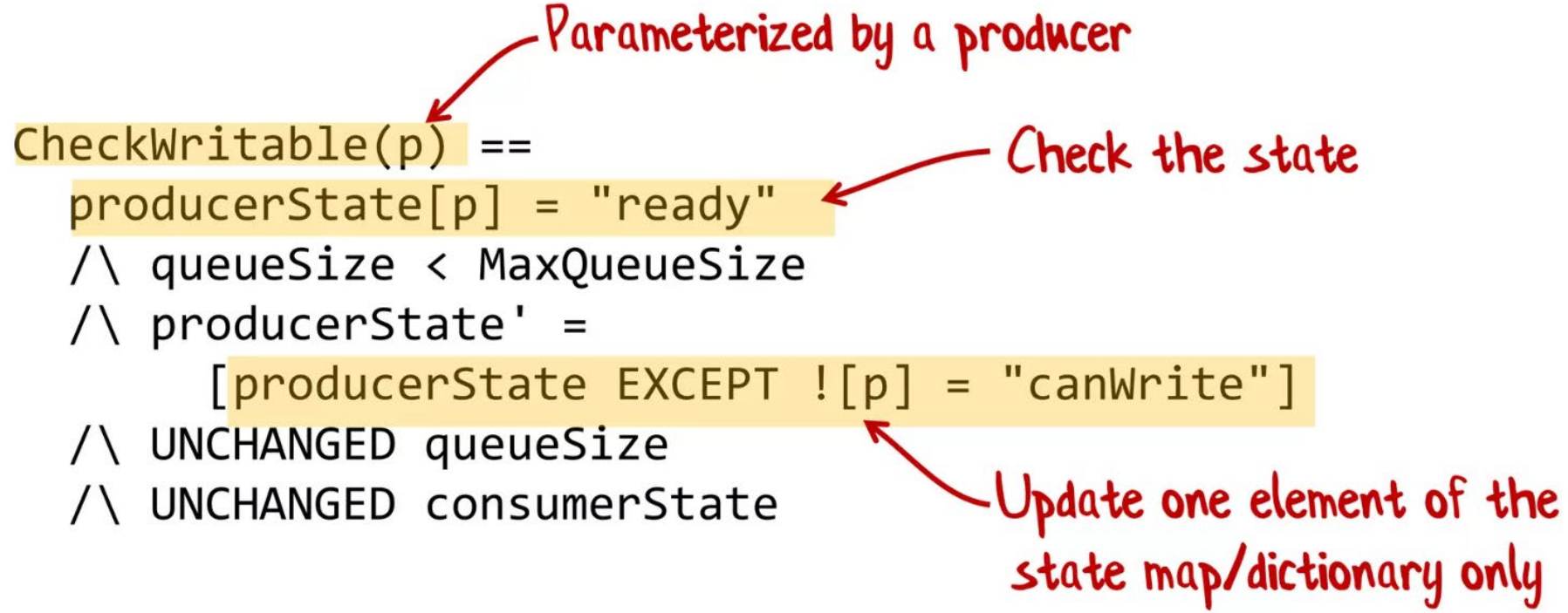
```
CONSTANT producers, consumers
\* e.g
\*    2 producers={"p1","p2"}
\*    2 consumers={"c1","c2"}

VARIABLES queueSize, producerState, consumerState

MaxQueueSize == 2

Init ==
  queueSize = 0
  /\ producerState = [p \in producers |-> "ready"]
                \* same as {"p1":"ready","p2":"ready"}
  /\ consumerState = [c \in consumers |-> "ready"]
```

These are now maps/dictionaries

For each producer, set the state to be "ready"

Producer/Consumer Spec, part 2

```
CheckWritable(p) ==
  producerState[p] = "ready"
  /\ queueSize < MaxQueueSize
  /\ producerState' =
       [producerState EXCEPT ![p] = "canWrite"]
  /\ UNCHANGED queueSize
  /\ UNCHANGED consumerState
```


Parameterized by a producer
CheckWritable(p) ==
producerState[p] = "ready"
Check the state
/\ queueSize < MaxQueueSize
/\ producerState' =
[producerState EXCEPT ![p] = "canWrite"]
/\ UNCHANGED queueSize
/\ UNCHANGED consumerState
Update one element of the state map/dictionary only

Producer/Consumer Spec, part 2

```
CheckWritable(p) ==
  producerState[p] = "ready"
  /\ queueSize < MaxQueueSize
  /\ producerState' =
       [producerState EXCEPT ![p] = "canWrite"]
  /\ UNCHANGED queueSize
  /\ UNCHANGED consumerState

Write(p) ==
  producerState[p] = "canWrite"
  /\ queueSize' =  queueSize + 1
  /\ producerState' =
       [producerState EXCEPT ![p] = "ready"]
  /\ UNCHANGED consumerState

ProducerAction ==
  \E p \in producers : CheckWritable(p) \/ Write(p)
```

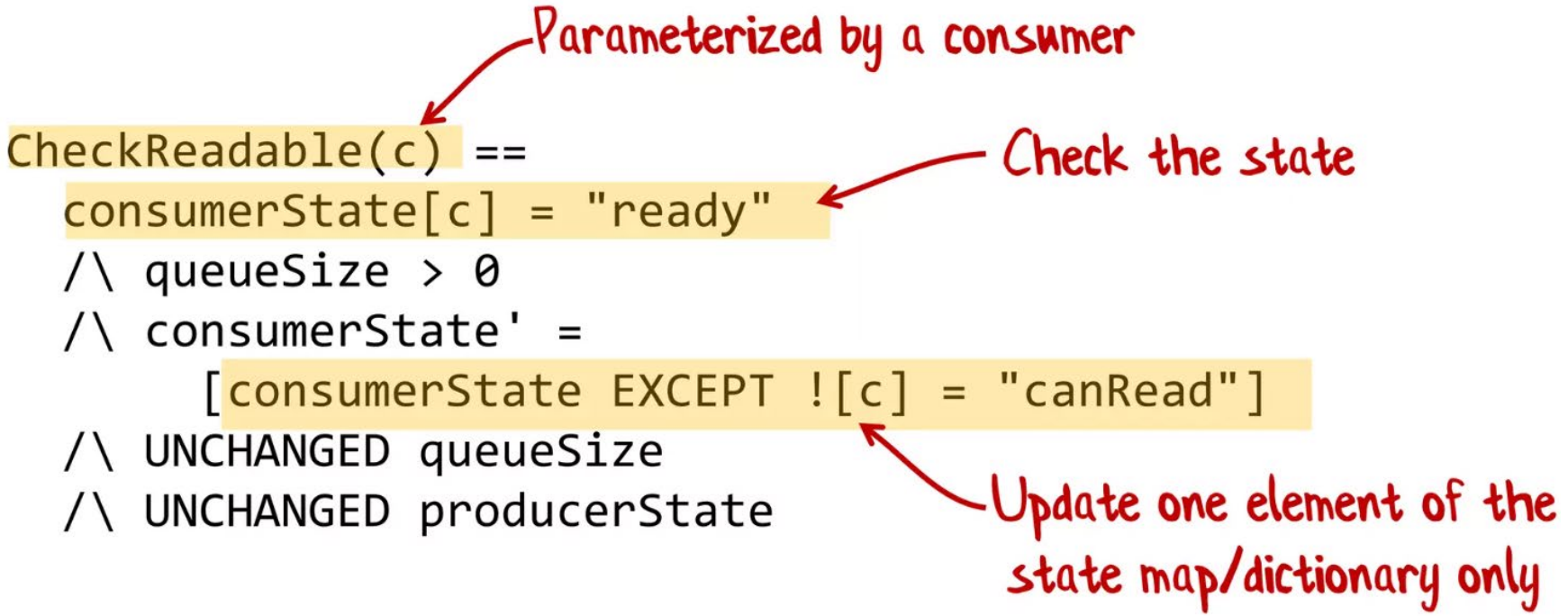
Producer/Consumer Spec, part 2

```
CheckWritable(p) ==
  producerState[p] = "ready"
  /\ queueSize < MaxQueueSize
  /\ producerState' =
       [producerState EXCEPT ![p] = "canWrite"]
  /\ UNCHANGED queueSize
  /\ UNCHANGED consumerState

Write(p) ==
  producerState[p] = "canWrite"
  /\ queueSize' =  queueSize + 1
  /\ producerState' =
       [producerState EXCEPT ![p] = "ready"]
  /\ UNCHANGED consumerState

ProducerAction ==
  \E p \in producers : CheckWritable(p) \/ Write(p)
```

Find any producer which has a valid action

Producer/Consumer Spec, part 3

```
CheckReadable(c) ==
  consumerState[c] = "ready"
  /\ queueSize > 0
  /\ consumerState' =
       [consumerState EXCEPT ![c] = "canRead"]
  /\ UNCHANGED queueSize
  /\ UNCHANGED producerState
```


Parameterized by a consumer
CheckReadable(c) ==
consumerState[c] = "ready"
/\ queueSize > 0
/\ consumerState' =
[consumerState EXCEPT ![c] = "canRead"]
/\ UNCHANGED queueSize
/\ UNCHANGED producerState
Check the state
Update one element of the state map/dictionary only

Producer/Consumer Spec, part 3

```
CheckReadable(c) ==
  consumerState[c] = "ready"
  /\ queueSize > 0
  /\ consumerState' =
       [consumerState EXCEPT ![c] = "canRead"]
  /\ UNCHANGED queueSize
  /\ UNCHANGED producerState

Read(c) ==
  consumerState[c] = "canRead"
  /\ queueSize' =  queueSize - 1
  /\ consumerState' =
       [consumerState EXCEPT ![c] = "ready"]
  /\ UNCHANGED producerState

ConsumerAction ==
  \E c \in consumers : CheckReadable(c) \/ Read(c)
```

```
CheckReadable(c) ==
  consumerState[c] = "ready"
  /\ queueSize > 0
  /\ consumerState' =
       [consumerState EXCEPT ![c] = "canRead"]
  /\ UNCHANGED queueSize
  /\ UNCHANGED producerState

Read(c) ==
  consumerState[c] = "canRead"
  /\ queueSize' =  queueSize - 1
  /\ consumerState' =
       [consumerState EXCEPT ![c] = "ready"]
  /\ UNCHANGED producerState

ConsumerAction ==
  \E c \in consumers : CheckReadable(c) \/ Read(c)
```

Find any consumer which has a valid action

Running the script

- Run the model checker with 2 producers and 2 consumers
  - Use the `AlwaysWithinBounds` property
- There are 38 states
- Error: `Invariant AlwaysWithinBounds is violated!`
  - The design does not work

Fixing the error

- TLA+ won’t tell you how to fix the error
  - You must fix the spec
- Easy to test the fixes
  - Update the spec to use atomic operations (or locks)
  - Re-run the model checker!
  - You gain confidence in your design

The power of TLA+

- TLA+ can be used to model large concurrent systems
  - Such as distributed systems!

- Examples where TLA+ can help:
  - https://hillelwayne.com/modeling-deployments/
  - https://hillelwayne.com/talks/distributed-systems-tlaplus/
- Learn more: https://learntla.com/index.html

Why concurrent programs are important?

- They are everywhere nowadays because we all use distributed systems
- Distributed systems use the most complex programs
  - systems that span the world
  - serve millions of users
  - and are always available
- Incredibly relevant today as everything is a distributed system!

Definition of Distributed Systems

- Distributed system is a system where multiple processes located on networked computers communicate via messages to achieve a common goal
- "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.“, Leslie Lamport
- Examples: client-server applications, map-reduce, grid computing, peer-to-peer networks, skype, cloud computing, email clients, music streaming, ftp connection, hadoop, web service compositions, video streaming, etc.

Definition of Distributed Systems

- Distributed system is a system where multiple processes located on networked computers communicate via messages to achieve a common goal
- "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.“, Leslie Lamport
- Examples: client-server applications, map-reduce, grid computing, peer-to-peer networks, skype, cloud computing, email clients, music streaming, ftp connection, hadoop, web service compositions, video streaming, etc.

Definition of Distributed Systems

- Distributed system is a system where multiple processes located on networked computers communicate via messages to achieve a common goal
- "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.“, Leslie Lamport
- Examples: client-server applications, map-reduce, grid computing, peer-to-peer networks, skype, cloud computing, email clients, music streaming, ftp connection, hadoop, web service compositions, video streaming, etc.

Challenges for distributed systems

- **Failure** – cope with partial failure - **reliability**
  - Hardware
  - Network
  - Software – distributed nature, but not only
  - Performance
- **No now**
  - No ordering, no centralized clock
- **Consistency**
- **Consensus**
- Security
- Testing

Major Challenges (1)

- No global clock, no ordering of events
  - Events happen at different times
  - Different interleaving of events are possible
  - The participants see the events interleaving in different ways
  - Use a logical clock (happens-before)

~ sounds like a memory model is needed

Major Challenges (2)

Resilience – Consistency - Consensus

Possible errors due to the concurrent nature of the system:

- Deadlocks
- Livelock/starvation
- Lack of consensus

~ knowing classical synchronization problems might help you solve some particular situation

2015: Formal Methods at AWS *

- Precise description of system in TLA+ (PlusCal language - like c)
  - In 6 large complex real world systems
  - 7 teams
- Found subtle bugs
- Confidence to make **aggressive optimizations** w/o sacrificing correctness
- Use formal specification **to teach** system to new engineers

* **How Amazon Web Services Uses Formal Methods** by Chris Newcombe et al. (Communications of the ACM, 2015)

2015: Formal Methods at AWB

Applying TLA+ to some of our more complex systems

System	Components	Line count (excl. comments)	Benefit
S3	Fault-tolerant low-level network algorithm	804 PlusCal	Found 2 bugs. Found further bugs in proposed optimizations.
	Background redistribution of data	645 PlusCal	Found 1 bug, and found a bug in the first proposed fix.
DynamoDB	Replication & group-membership system	939 TLA+	Found 3 bugs, some requiring traces of 35 steps
EBS	Volume management	102 PlusCal	Found 3 bugs.
Internal distributed lock manager	Lock-free data structure	223 PlusCal	Improved confidence. Failed to find a liveness bug as we did not check liveness.
	Fault tolerant replication and reconfiguration algorithm	318 TLA+	Found 1 bug. Verified an aggressive optimization.

2021: Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3

- S3's new ShardStore storage node
  - Built in Rust
  - Crash consistency, concurrency, IO,etc
- Specs alongside the code
  - Reference model spec
  - Decompose correctness checks
    - Sequential correctness
    - Crashes
    - Concurrency
  - Accept weaker correctness guarantees then full formal verification
- Adding continuous validation validation

* Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3 by James Bornholt et al., SOSP2021. https://www.youtube.com/watch?v=YdxvOPenjWI

Conclusion

- Formal verification (model checking) bring guarantees and allows us to check properties
- Formally checking the concurrent code is here to stay
- More engineers will need to write formal specs for their code
- Industry is adapting and using model checkers, especially for newly developed systems

- References:
  - https://www.youtube.com/watch?v=tqwcz-Yt9gQ