TLA+ Tiramisu

Hillel Wayne

Levels of Expertise

- Beginner
- Intermediate
- Expert

Levels of Expertise

- Beginner (learntla, SpecifyingSystems, Practical TLA+)
- Intermediate
- Expert

Levels of Expertise

- Beginner (learntla, SpecifyingSystems, Practical TLA+)
- Intermediate ???
- Expert

www.learntla.com

Resuming in 2022!

www.learntla.com

Resuming in 2022!

- Rewrite as Sphinx

www.learntla.com

Resuming in 2022!

- Rewrite as Sphinx
- Clean up beginner content

www.learntla.com

Resuming in 2022!

- Rewrite as Sphinx
- Clean up beginner content
- More intermediate content


Conjunction
Capers
A TLA+ Truffle
Ron Pressler, October 2020
0:00 / 27:24

What I *wanted* to cover

1. Action Properties
2. Complex Liveness Properties
3. Modules
4. Fairness and machine closure
5. Aux vars
6. TLC configuration
7. Model Optimization
8. Community Modules

Actual Outline

1. Action Properties
2. Modules
3. Model Optimization
4. Miscellaneous

Note

I might jump between tla+ and pluscal

Action Properties

Invariants and Temporal Properties

Safety	Bad things don't happen
Liveness	Good things happen

Invariants and Temporal Properties

Safety	Bad things don't happen
Liveness	Good things happen
Invariants	Props on states
Properties	Props on behaviors

Action Properties

Properties over actions

Action Properties

Properties over actions

```
x' >= x
```

Action Properties

Properties over actions

```
[x' >= x]_x
```

Action Properties

Properties over actions

```
[][x' >= x]_x
```

Action Properties

Properties over actions

```
XAlwaysIncrements ==
  [][x' >= x]_x
```

Best Practices

1. `[][x' >= x]_x`
2. `[][x' > x]_x`

TLC can only compute

```
x' = expr
x' \in set
```

TLC can only compute

```
x' = expr
x' \in set
```

TLC can check any action

TLC can only compute

```
x' = expr
x' \in set
```

TLC can check any action

```
[][SubSeq(log', 1, Len(log)) = log]_log
```

Conditionals

```
[][action => expr]_var
```

Conditionals

```
[][action => expr]_var
```

`action` can only happen if `expr` is true

Conditionals

```
[][action => expr]_var
```

`action` can only happen if `expr` is true

```
[][x' # x => enabled]_<<x, enabled>>
```

Conditionals

```
[][action => expr]_var
```

`action` can only happen if `expr` is true

```
[][x' # x => enabled]_<<x, enabled>>
```

```
[][x # NULL => x' # NULL]_x
[][x # NULL => x' = x]_x
```

Action action properties

```
Next == A \/ B
```

Action action properties

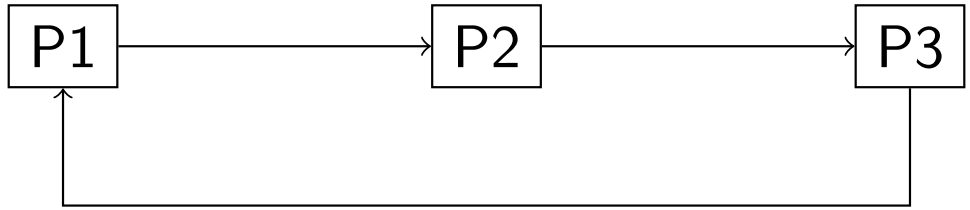
```
Next == A \/ B

[][A => enabled]_vars
```

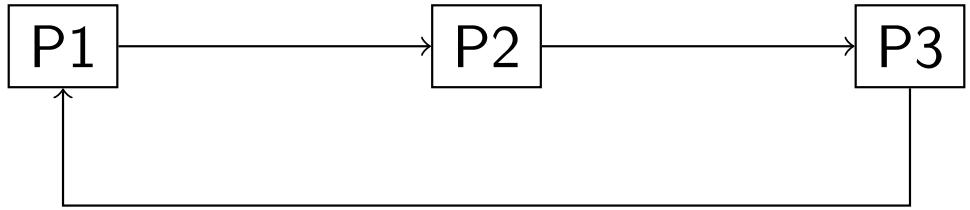
Action action properties

```
Next == A \/ B

[][A => enabled]_vars

[][enabled => A]_vars
```

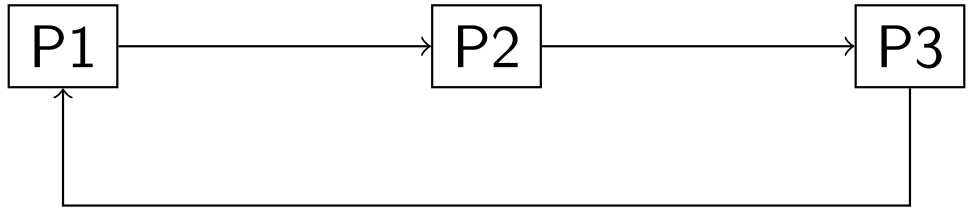
Action action properties

```
Next == A \/ B

[][A => enabled]_vars

[][enabled => A]_vars

[][action1 => action2]_vars
```


P1
P2
P3

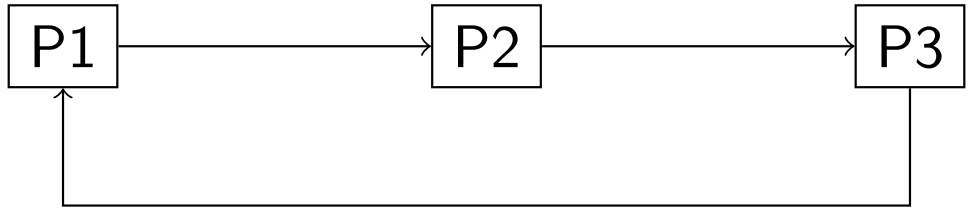



```
[][
  /\ pc = "P1" => pc' \in {"P1", "P2"}
  /\ pc = "P2" => pc' \in {"P2", "P3"}
  /\ pc = "P3" => pc' \in {"P3", "P1"}
]_pc
```




```
[][
  /\ pc = "P1" => pc' \in {"P1", "P2"}
  /\ pc = "P2" => pc' \in {"P2", "P3"}
  /\ pc = "P3" => pc' \in {"P3", "P1"}
]_pc
```

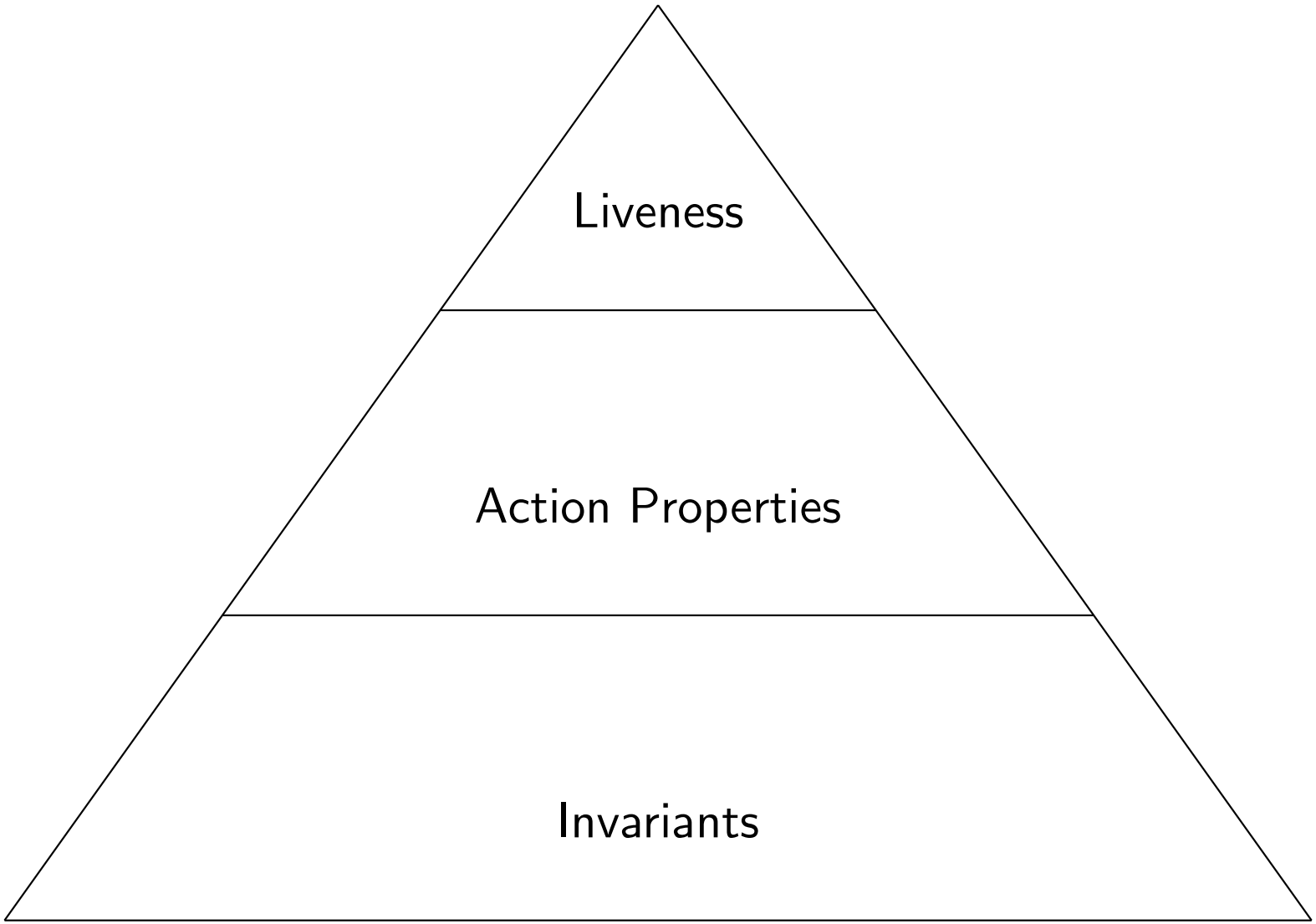
```
[][<<pc, pc'>> \in {
  <<"P1", "P1">>, <<"P1", "P2">>,
  <<"P2", "P2">>, <<"P2", "P3">>,
  <<"P3", "P3">>, <<"P3", "P1">>
}]_pc
```




```
[][
  /\ pc = "P1" => pc' \in {"P1", "P2"}
  /\ pc = "P2" => pc' \in {"P2", "P3"}
  /\ pc = "P3" => pc' \in {"P3", "P1"}
]_pc
```

```
[][<<pc, pc'>> \in {
  <<"P1", "P1">>, <<"P1", "P2">>,
  <<"P2", "P2">>, <<"P2", "P3">>,
  <<"P3", "P3">>, <<"P3", "P1">>
}]_pc
```

```
  [][<<pc, pc'>> = <<1, 2>
      => y' = y+1]_vars
```


Liveness
Action Properties
Invariants

https://www.hillelwayne.com/post/action-properties/

Modules

Stack Ops

```
Push(stack, x) == Append(stack, x)

Pop(stack) == SubSeq(stack, 1, Len(stack)-1)
```

Stack Ops

```
---- MODULE stack ----

LOCAL INSTANCE Sequences
LOCAL INSTANCE Integers

Push(stack, x) == Append(stack, x)

Pop(stack) == SubSeq(stack, 1, Len(stack)-1)

====
```

Stack Ops

```
---- MODULE stack ----

LOCAL INSTANCE Sequences
LOCAL INSTANCE Integers

Push(stack, x) == Append(stack, x)

Pop(stack) == SubSeq(stack, 1, Len(stack)-1)

====
```

Importing

```
EXTENDS Stack
```

Importing

```
EXTENDS Stack
```

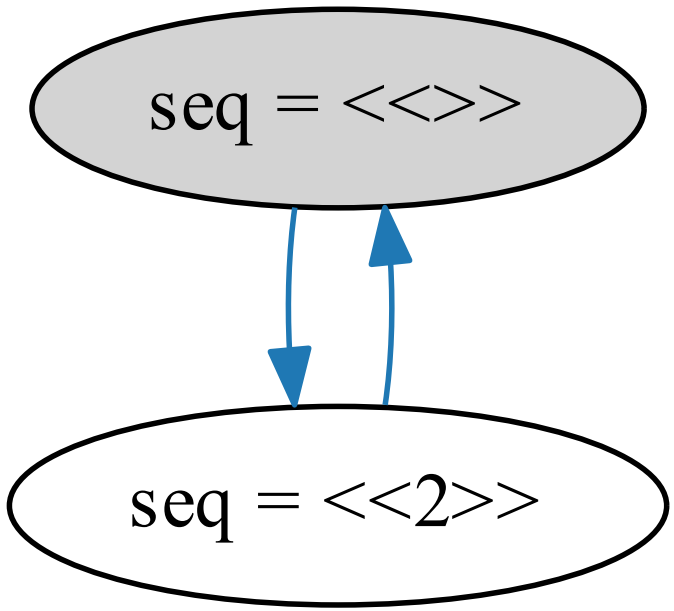
```
Stack == INSTANCE stack
```

Importing

```
EXTENDS Stack


Stack == INSTANCE stack

Stack!Push
```

main.tla

```
---- MODULE main ----
EXTENDS Integers, Sequences, TLC
Stack == INSTANCE stack

VARIABLES seq

Init == seq = <<>>

Next ==
  \/ /\ seq = <<>>
     /\ seq' = Stack!Push(seq, 2)
  \/ /\ seq # <<>>
     /\ seq' = Stack!Pop(seq)

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


seq = <<>>
seq = <<2>>

Action Modules

Stack.tla

```
Push(stack, x) ==
  stack' = Append(stack, x)

Pop(stack) ==
  stack' = SubSeq(stack, 1, Len(stack)-1)
```

Action Modules

Stack.tla

```
Push(stack, x) ==
  stack' = Append(stack, x)

Pop(stack) ==
  stack' = SubSeq(stack, 1, Len(stack)-1)
```

main.tla

```
\/ /\ seq = <<>>
   /\ Stack!Push(seq, 2)
\/ /\ seq # <<>>
   /\ Stack!Pop(seq)

\* ^^ was seq' = Stack!Pop(seq)
```

Stack.tla

```
VARIABLE stack

Push(x) ==
  stack' = Append(stack, x)

Pop ==
  stack' = SubSeq(stack, 1, Len(stack)-1)
```

Stack.tla

```
VARIABLE stack

Push(x) ==
  stack' = Append(stack, x)

Pop ==
  stack' = SubSeq(stack, 1, Len(stack)-1)
```

Main.tla

```
Stack == INSTANCE stack WITH stack <- seq

Next ==
\/ /\ seq = <<>>
    /\ Stack!Push(2)
\/ /\ seq # <<>>
    /\ Stack!Pop
```

Stack.tla

```
Next ==
\/ Pop
\/ \E x \in 1..2:
    Push(x)

Spec == [][Next]_stack
```

Stack.tla

```
Next ==
\/ Pop
\/ \E x \in 1..2:
    Push(x)

Spec == [][Next]_stack
```

Main.tla

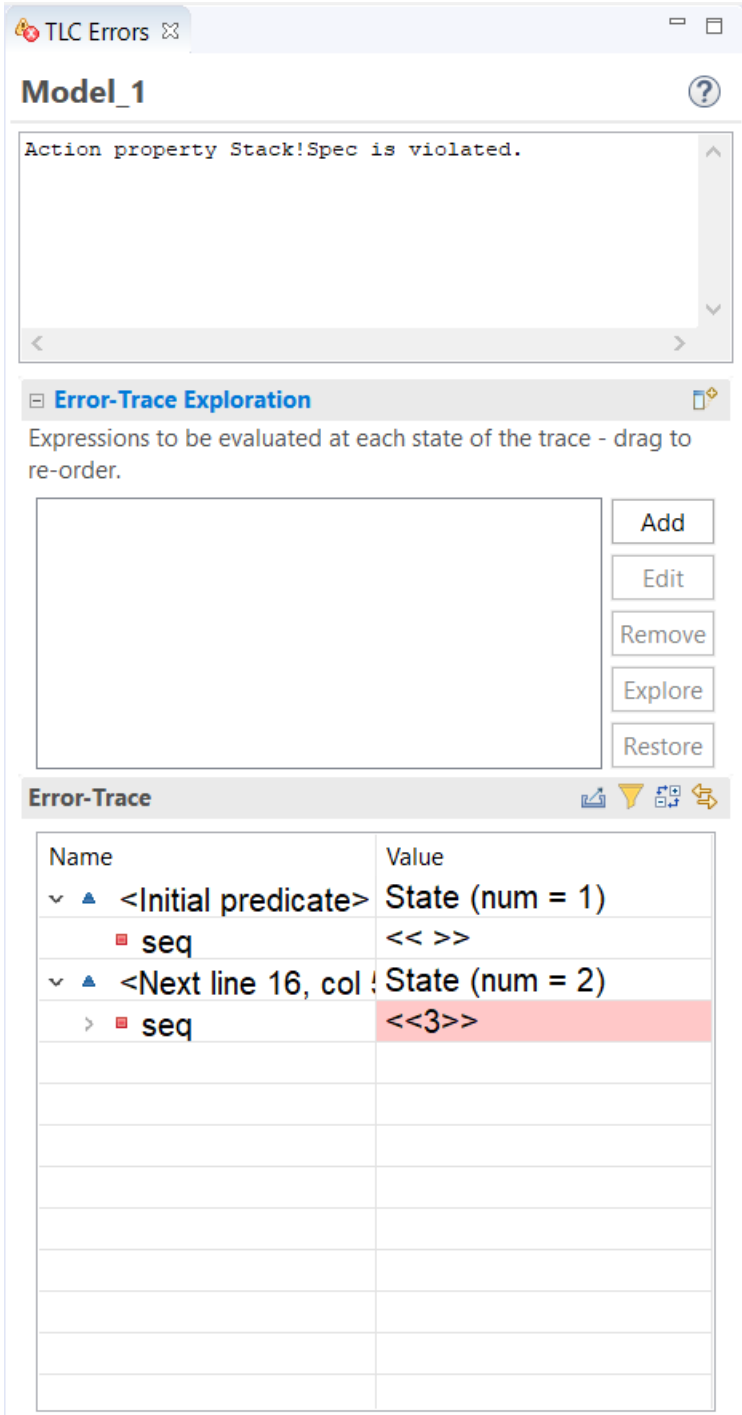
```
Refinement == Stack!Spec
```

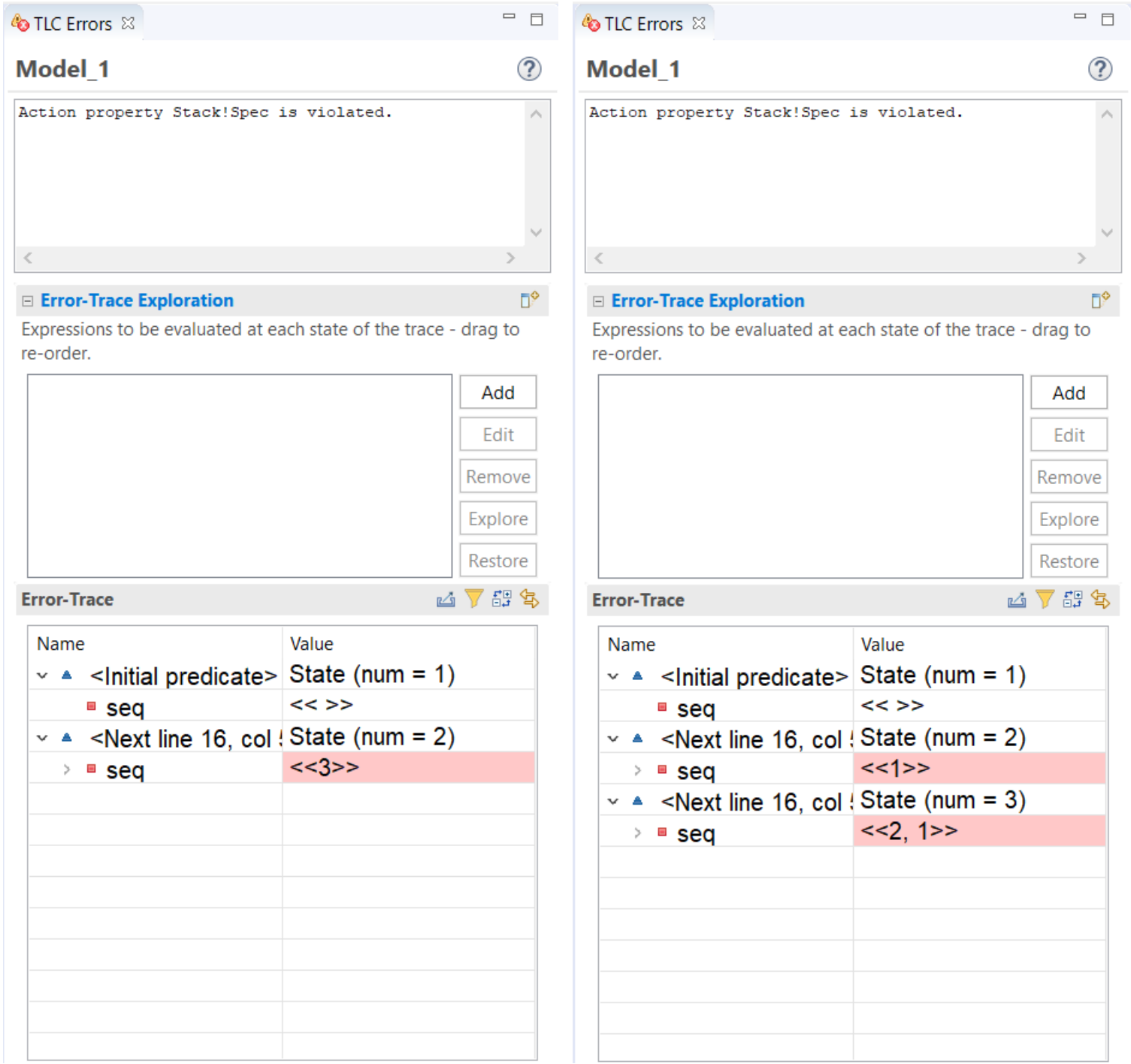
Stack.tla

```
Next ==
\/ Pop
\/ \E x \in 1..2:
    Push(x)

Spec == [][Next]_stack
```

Main.tla

```
Refinement == Stack!Spec

\*  [][Stack!Next]_seq
```


TLC Errors
Model_1
Action property Stack!Spec is violated.
Error-Trace Exploration
Expressions to be evaluated at each state of the trace - drag to re-order.
Add
Edit
Remove
Explore
Restore
Error-Trace
Name
Value
<Initial predicate>
State (num = 1)
seq
<< >>
<Next line 16, col
State (num = 2)
seq
<<3>>


TLC Errors
Model_1
Action property Stack!Spec is violated.
Error-Trace Exploration
Expressions to be evaluated at each state of the trace - drag to re-order.
Add
Edit
Remove
Explore
Restore
Error-Trace
Name
Value
<Initial predicate>
State (num = 1)
seq
<< >>
<Next line 16, col
State (num = 2)
seq
<<3>>
TLC Errors
Model_1
Action property Stack!Spec is violated.
Error-Trace Exploration
Expressions to be evaluated at each state of the trace - drag to re-order.
Add
Edit
Remove
Explore
Restore
Error-Trace
Name
Value
<Initial predicate>
State (num = 1)
seq
<< >>
<Next line 16, col
State (num = 2)
seq
<<1>>
<Next line 16, col
State (num = 3)
seq
<<2, 1>>

Stack.tla

```
CONSTANT StackType
Next ==
  Pop \/ \E x \in StackType:
     Push(x)
```

Stack.tla

```
CONSTANT StackType
Next ==
  Pop \/ \E x \in StackType:
     Push(x)
```

Main.tla

```
Workers == {"a", "b"}
Stack == INSTANCE stack WITH stack <- seq,
   StackType <- Workers
Range(f) == {f[x] : x \in DOMAIN f}

Next ==
  \/ /\ seq # <<>>
     /\ Stack!Pop
  \/ \E w \in Workers:
    /\ w \notin Range(seq)
    /\ Stack!Push(w)
```

Stack.tla

```
Init == stack \in Seq(StackType)
Spec == Init /\ [][Next]_stack
```

Detail 1

```
Refinement == Stack!Spec
```

Detail 1

```
THEOREM Spec => Stack!Spec
```

Detail 2

```
Stack == INSTANCE Stack WITH stack <- seq
```

Detail 2

```
Stack == INSTANCE Stack WITH stack <- seq

Stack == INSTANCE Stack WITH stack <- f.q \o f.p
```

Refinement

Verifying a low-level spec implements a high-level spec, usually deterministically

Refinement

Verifying a low-level spec implements a high-level spec, usually deterministically

```
\* abstract
balance \in Int

\* concrete
transactions \in Seq(Int)
```

Refinement

Verifying a low-level spec implements a high-level spec, usually deterministically

```
\* abstract
balance \in Int

\* concrete
transactions \in Seq(Int)

Acct == INSTANCE BankAcct
        WITH balance <- SumSeq(transactions)
```

https://www.hillelwayne.com/post/refinement/

Optimization

Unbound Models

Where there's an infinite number of reachable states.

Bound Models

1. $TimePerAction \times NumStates$

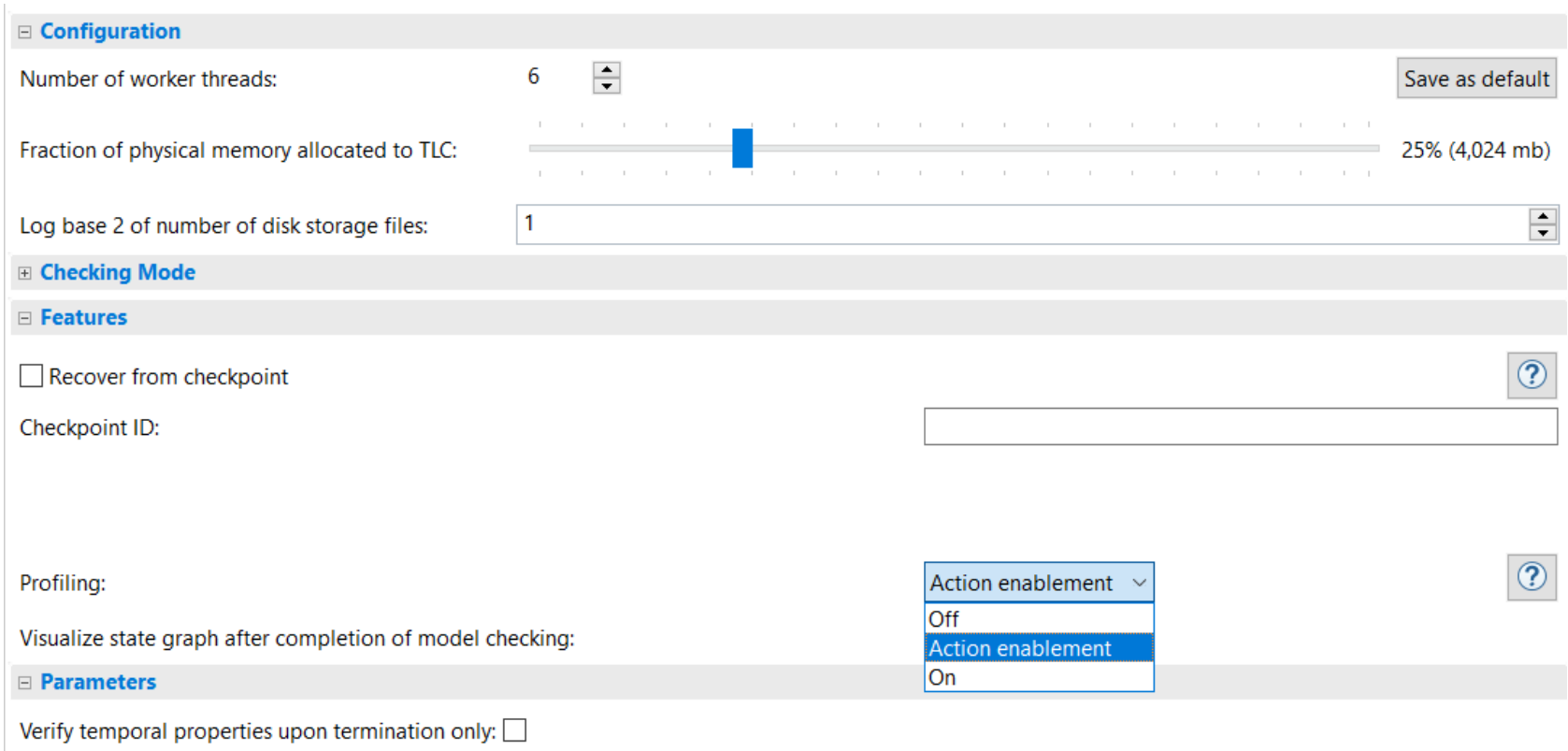
Bound Models

1. $TimePerAction \times NumStates$
2. Slow Actions
3. Lots of States

Bound Models

1. $TimePerAction \times NumStates$
2. Slow Actions
3. Lots of States

Profiler


Configuration
Number of worker threads:
6
Save as default
Fraction of physical memory allocated to TLC:
25% (4,024 mb)
Log base 2 of number of disk storage files:
1
Checking Mode
Features
Recover from checkpoint
Checkpoint ID:
Profiling:
Action enablement
Off
Action enablement
On
Visualize state graph after completion of model checking:
Parameters
Verify temporal properties upon termination only:

Profiler

```
32
33 end algorithm;*)
34 \* BEGIN TRANSLATION - the hash of the PCal code: PCal
35 VARIABLES seq, pc
36
37 vars == << seq, pc >>
38
39 ProcSet == {1} \cup {2}
40
41 Init == (* Global variables *)
42         /\ seq = <<>>
43         /\ pc = [self \in ProcSet |-> CASE self = 1 ->
44                                         [] self = 2 ->
45
46 A == /\ pc[1] = "A"
47    /\ seq' = Append(seq, "A1")
48    /\ pc' = [pc EXCEPT ![1] = "B"]
49
50 B == /\ pc[1] = "B"
51    /\ seq' = Append(seq, "A2")
52    /\ pc' = [pc EXCEPT ![1] = "C"]
53
```

Set sizes

```
\* |S| == Cardinality(S)

1. |SUBSET S| = 2^|S|
2. |S \X T|   = |S|*|T|
3. |[S -> T]| = |T|^|S|
```

How many initial states?

```
variable
  network \in [Server -> SUBSET Client]
```

How many initial states?

```
variable
  network \in [Server -> SUBSET Client]
```

$(2^{|C|})^{|S|}$

How many initial states?

```
variable
  network \in [Server -> SUBSET Client]
```

$(2^{|C|})^{|S|}$

For 3 servers & 3 clients, that's 512 networks

How many next states?

```
SpiteMe:
  with n \in [Server -> SUBSET Client] do
    network := n;
  end with;
```

How many next states?

```
SpiteMe:
  with n \in [Server -> SUBSET Client] do
    network := n;
  end with;
```

512 new states *each time*

Grain of Atomicity

The less you change per action, the more concurrency you have.

"Cut points"

This is bad

```
i := 0;
Label:
  while i <= 5 do
    seq := Append(seq, i*2);
    i += 1;
  end while;
```

This is bad

```
i := 0;
Label:
  while i <= 5 do
    seq := Append(seq, i*2);
    i += 1;
  end while;
```

Should just be

```
  seq := seq \o [x \in 0..5 |-> x*2]
```

Minimize Distinct States

Do you *need* ordering for your data? Use a bag.

Sources of Unbound Models

```
\* common
i := i + 1
seq := Append(seq, n)

\* less common
f' = f @@ (a :> b)
seq' = <<seq>>
```

Use a state constraint!

State Constraints

```
TypeInvariant ==
  /\ i \in Nat
  /\ seq \in Seq(Nat)
```

State Constraints

```
TypeInvariant ==
  /\ i \in Nat
  /\ seq \in Seq(Nat)

ModelStateConstraint ==
  /\ i \in 0..MaxInt
  /\ seq \in Seq(Nat)
  /\ Len(seq) <= 5
```

Warning

You will (probably) lose liveness

TLCGet

TLCGet(“Level”)

Additional Definitions

Definitions required for the model checking, in addition to the definitions in the specification modules.

Definition Override

Directs TLC to use alternate definitions for operators.

Add

Edit

Remove

Model Values

An additional set of model values.

Action Constraint

A formula restricting a transition if its evaluation is not satisfied.

State Constraint

A formula restricting the possible states by a state predicate.

```
TLCGet("level") <= 4
```

Miscellaneous

@

Small thing for functions

```
f' = [f EXCEPT f[a].b[3] = f[a].b[3] + 1]
f' = [f EXCEPT f[a].b[3] = @ + 1]
```

Small thing, for deep function updates

Function Decomposition

Instead of

```
WorkerState == [queue: Seq(Msg), online: BOOLEAN]
variables
  state \in [Worker -> WorkerState];
```

Do

```
variables
  worker_queue \in [Worker -> Seq(Msg)];
  worker_online \in [Worker -> BOOLEAN];
```

Top-level actions

Instead of

```
Add ==
  \E w \in Worker: s' = s \union {w}
Remove ==
  \E w \in Worker: s' = s \ {w}

Next == Add \/ Remove
```

Top-level actions

Instead of

```
Add ==
  \E w \in Worker: s' = s \union {w}
Remove ==
  \E w \in Worker: s' = s \ {w}

Next == Add \/ Remove
```

Do

```
Add(w) == s' = s \union {w}
Remove(w) == s' = s \ {w}

Next ==
  \E w \in Worker:
    Add(w) \/ Remove(w)
```

State Sweeping

Instead of

```
CONSTANT Workers
ASSUME Workers > 1

variables workers \in 1..Workers
```

State Sweeping

Instead of

```
CONSTANT Workers
ASSUME Workers > 1

variables workers \in 1..Workers
```

Try

```
CONSTANT MaxWorkers
ASSUME MaxWorkers > 1

variables
  W \in 1..MaxWorkers;
  Workers \in 1..W; \* Sweep!
```

```
  W \in 1..MaxWorkers;
  w_status \in [1..W -> STATUS];

define
  W_is_static == [][W' = W]_W
end define;
```

Community Modules!

https://github.com/tlaplus/CommunityModules/