Extending Apalache to Symbolically Reason about Temporal Properties of TLA+

TLA+ Conference
22.9.2022

Philip Offtermatt

Joint work with Jure Kukovec and Igor Konnov

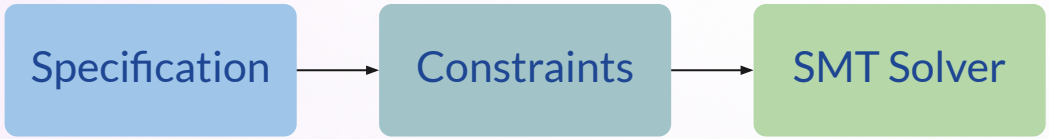
informal SYSTEMS

What is Apalache?

Apalache

Symbolic model checker for TLA+ – formally verify TLA+ specifications for real-world distributed systems protocols

- **Symbolic bounded model checker**
  vs TLC: state-space enumeration



- **Symbolically reason about infinite state-spaces**
  "amount \in Nat"

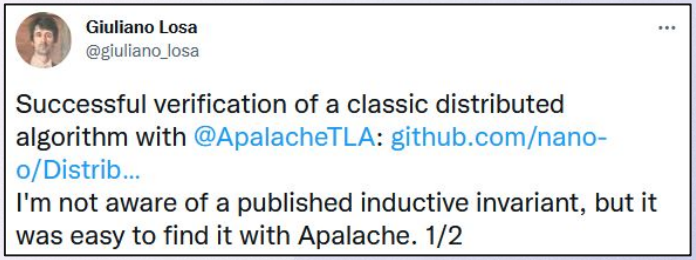
- **Bounded executions**
  "There is no invariant violation in the first 50 steps"

- **Adds Types and Type Checking to TLA+**

- **Successfully used to verify Tendermint**

- **Also used by external users to verify other distributed algorithms**

```
VARIABLE
  \* @type: Int;
  amount
```



| What is Apalache?

Developed at informal SYSTEMS

Team:

Igor Konnov | Jure Kukovec | Shon Feder

Rodrigo Otoni | Gabriela Moreira

Thomas Pani | Philip Offtermatt (Internship)

Releases, Manual, Tutorials, Example Specs, …

https://apalache.informal.systems/

| Apalache keeps you safe…
…but lacks support for liveness

Apalache

Symbolic model checker for TLA+ – formally verify TLA+ specifications for real-world distributed systems protocols

state invariants
"balances never go negative"

```
StateInvariant == balance >= 0
```

action invariants
"each round inflates the token supply by exactly 200"

```
ActionInvariant ==
    tokens' = tokens + 200
```

trace invariants
"the token supply in the last state is twice as large as in the first state"

```
TraceInvariant(hist) ==
    hist[Len(hist)].tokens =
        hist[1].tokens * 2
```

Liveness?

"eventually something good happens"

Trace invariants:

- can express liveness
- are hard to write

```
Liveness == <> tokens >= 2
```

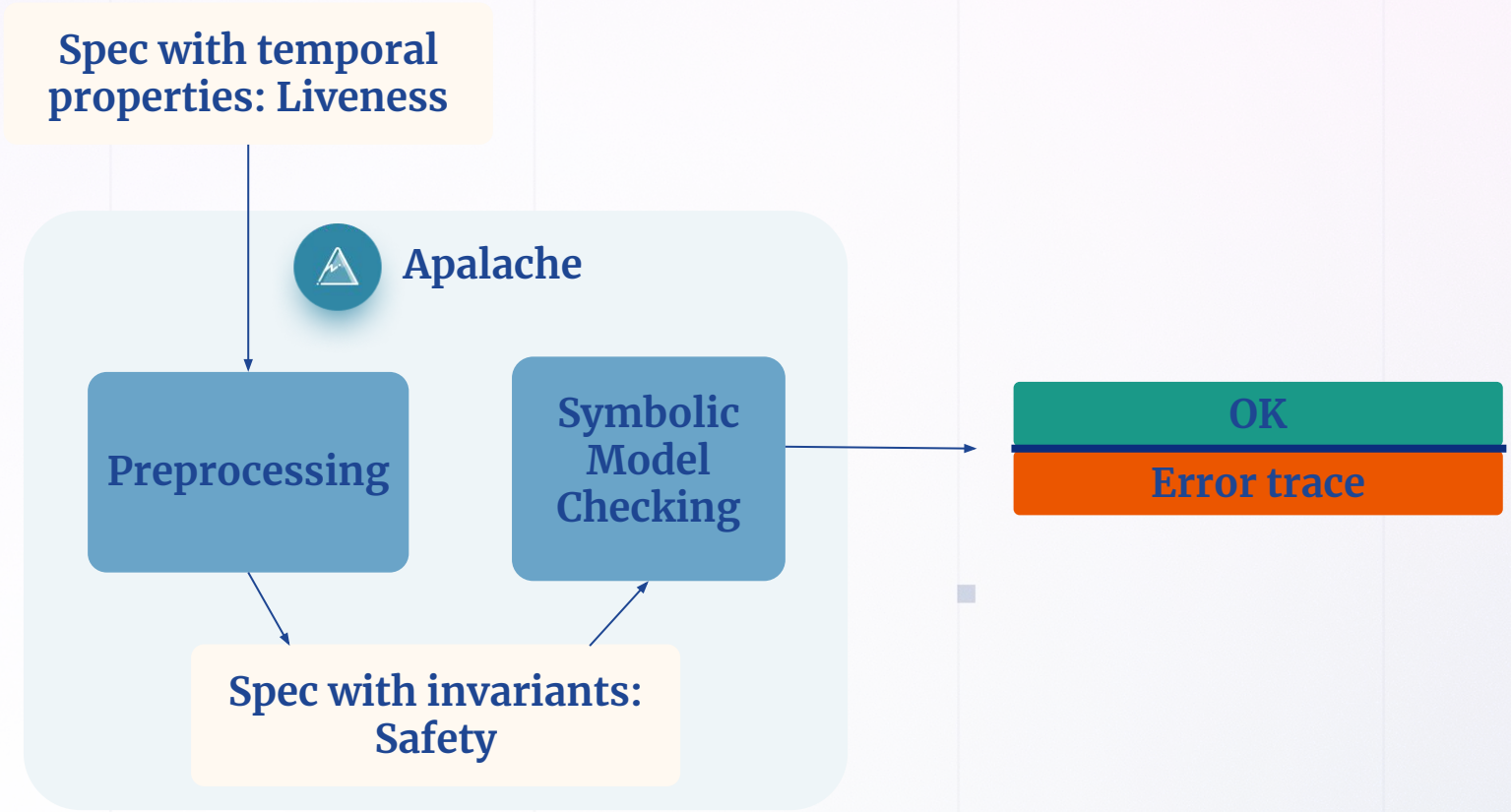
vs.

```
Liveness(hist) ==
    \E step \in DOMAIN hist:
        hist[step].tokens >= 2
```

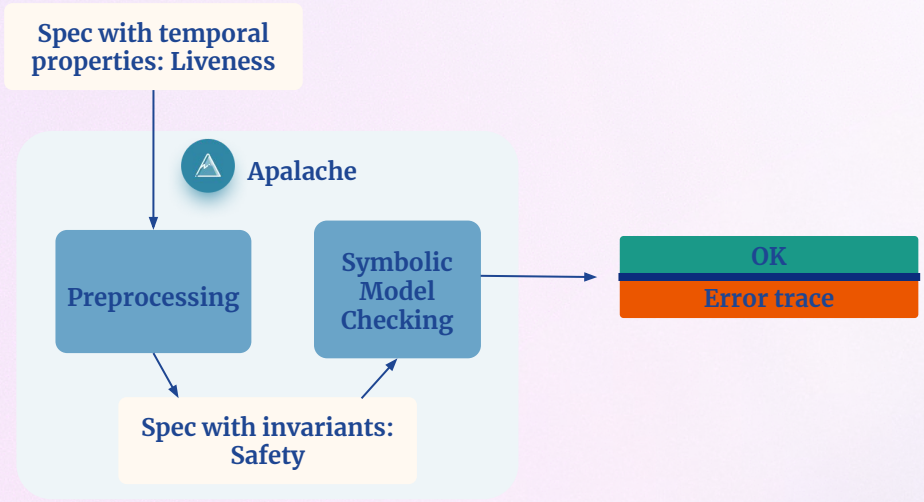
Goal:

native support for temporal properties in Apalache

Liveness-To-Safety



What can you get out of this talk?



(1) What are counterexamples to liveness?

(2) How to transform liveness to safety properties

(3) Tricks for applying techniques for classical LTL to TLA+: History/Prophecy Variables

Biere et al.: Linear Encodings of Bounded LTL Model Checking

Logical Methods in Computer Science
Vol. 2 (5:5) 2006, pp. 1–64
www.lmcs-online.org

Submitted Feb. 16, 2006
Published Nov. 15, 2006

LINEAR ENCODINGS OF BOUNDED LTL MODEL CHECKING

ARMIN BIERE [a], KEIJO HELJANKO [b], TOMMI JUNTTILA [c], TIMO LATVALA [d], AND VIKTOR SCHUPPAN [e]

[a] Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstrasse 69, A-4040 Linz, Austria
e-mail address: biere@jku.at

[b,c] Laboratory for Theoretical Computer Science, Helsinki University of Technology, P.O. Box 5400, FI-02015 TKK, Finland
e-mail address: {Keijo.Heljanko,Tommi.Junttila}@tkk.fi

[d] Department of Computer Science, University of Illinois at Urbana-Champaign, 201 Goodwin Ave., Urbana, IL 61801-2302, USA
e-mail address: tlatvala@uiuc.edu

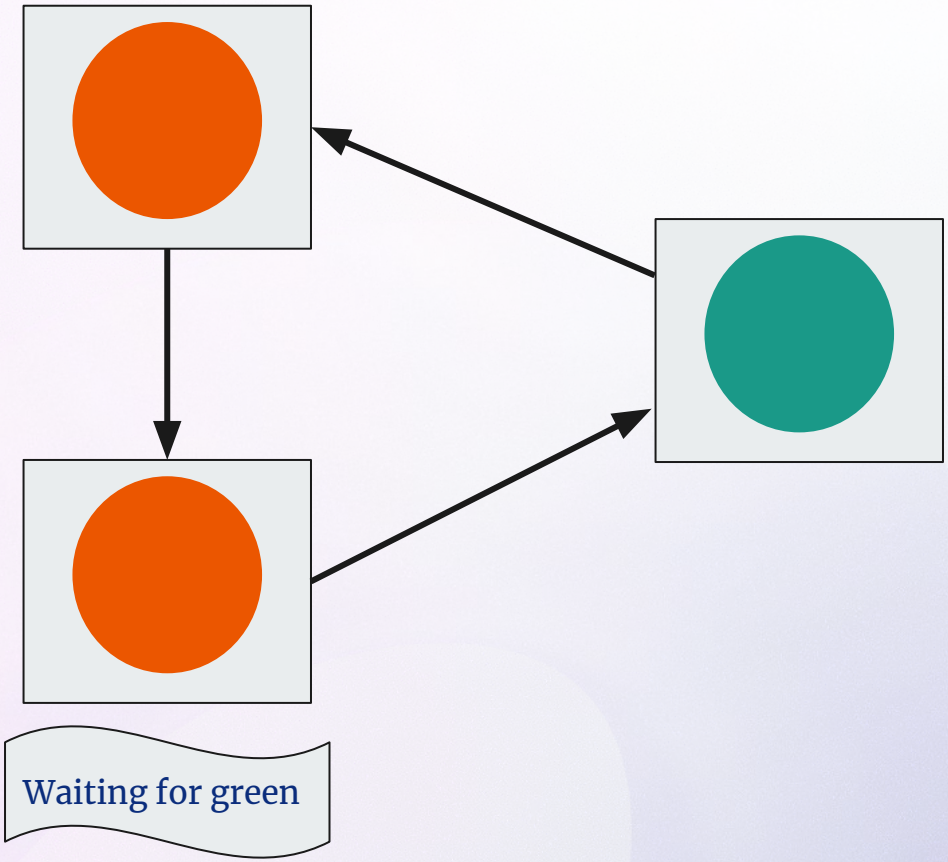
[e] Computer Systems Institute, ETH Zentrum, CH-8092 Zürich, Switzerland
e-mail address: vschuppan@acm.org

ABSTRACT. We consider the problem of bounded model checking (BMC) for linear temporal logic (LTL). We present several efficient encodings that have size linear in the bound. Furthermore, we show how the encodings can be extended to LTL with past operators (PLTL). The generalised encoding is still of linear size, but cannot detect minimal length counterexamples. By using the virtual unrolling technique minimal length counterexamples can be captured, however, the size of the encoding is quadratic in the specification. We also extend virtual unrolling to Büchi automata, enabling them to accept minimal length counterexamples.

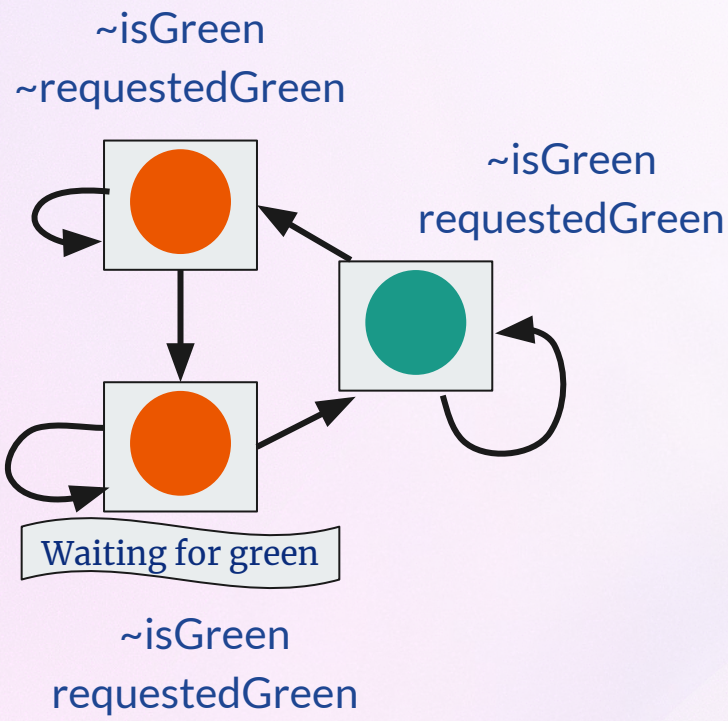
Our BMC encodings can be made incremental in order to benefit from incremental SAT technology. With fairly small modifications the incremental encoding can be further enhanced with a termination check, allowing us to prove properties with BMC.

An analysis of the liveness-to-safety transformation reveals many similarities to the BMC encodings in this paper. We conduct experiments to determine the advantage of employing dedicated BMC encodings for PLTL over combining more general but potentially less efficient approaches with BMC: the liveness-to-safety transformation with invariant

A toy example: TrafficLight



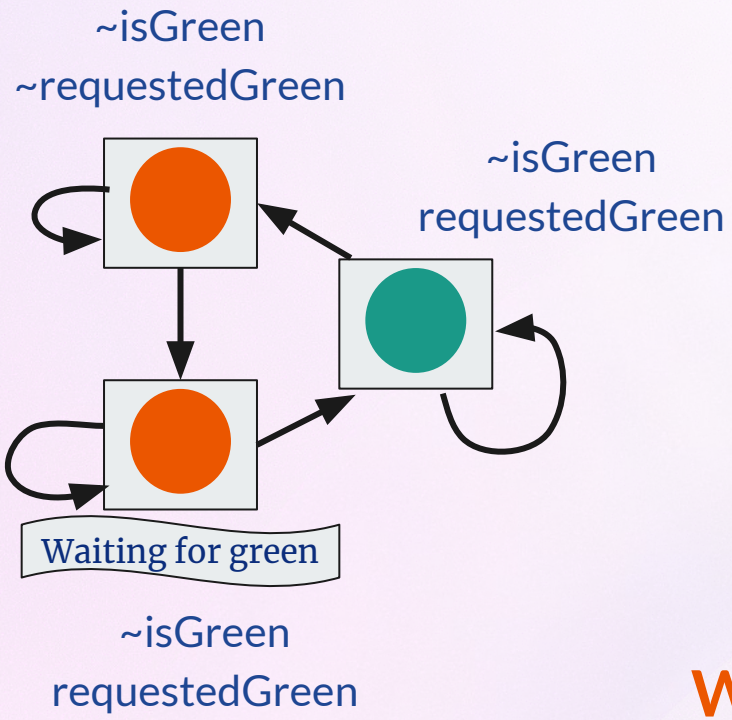
A specification for the TrafficLight




```
VARIABLES
        \* @type: Bool;
        isGreen,

        \* @type: Bool;
        requestedGreen
```

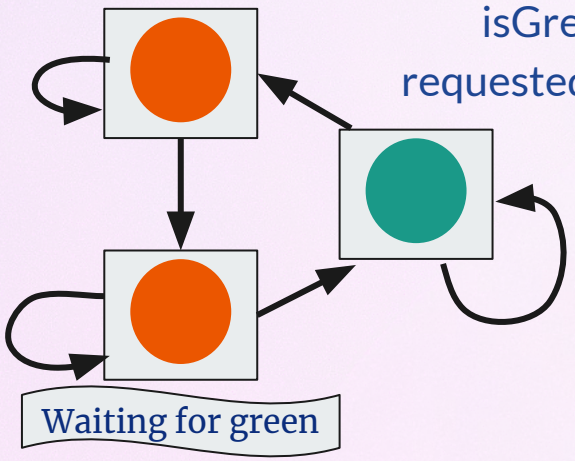
A liveness property for the TrafficLight

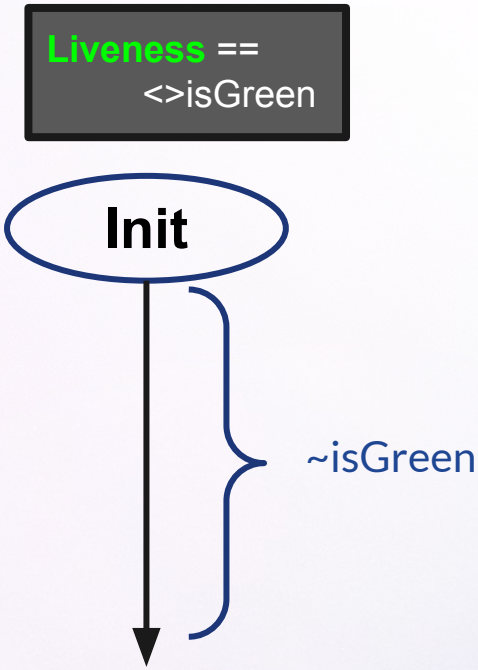


Liveness == <>isGreen

What do violations to liveness properties look like?

Counterexamples to Liveness

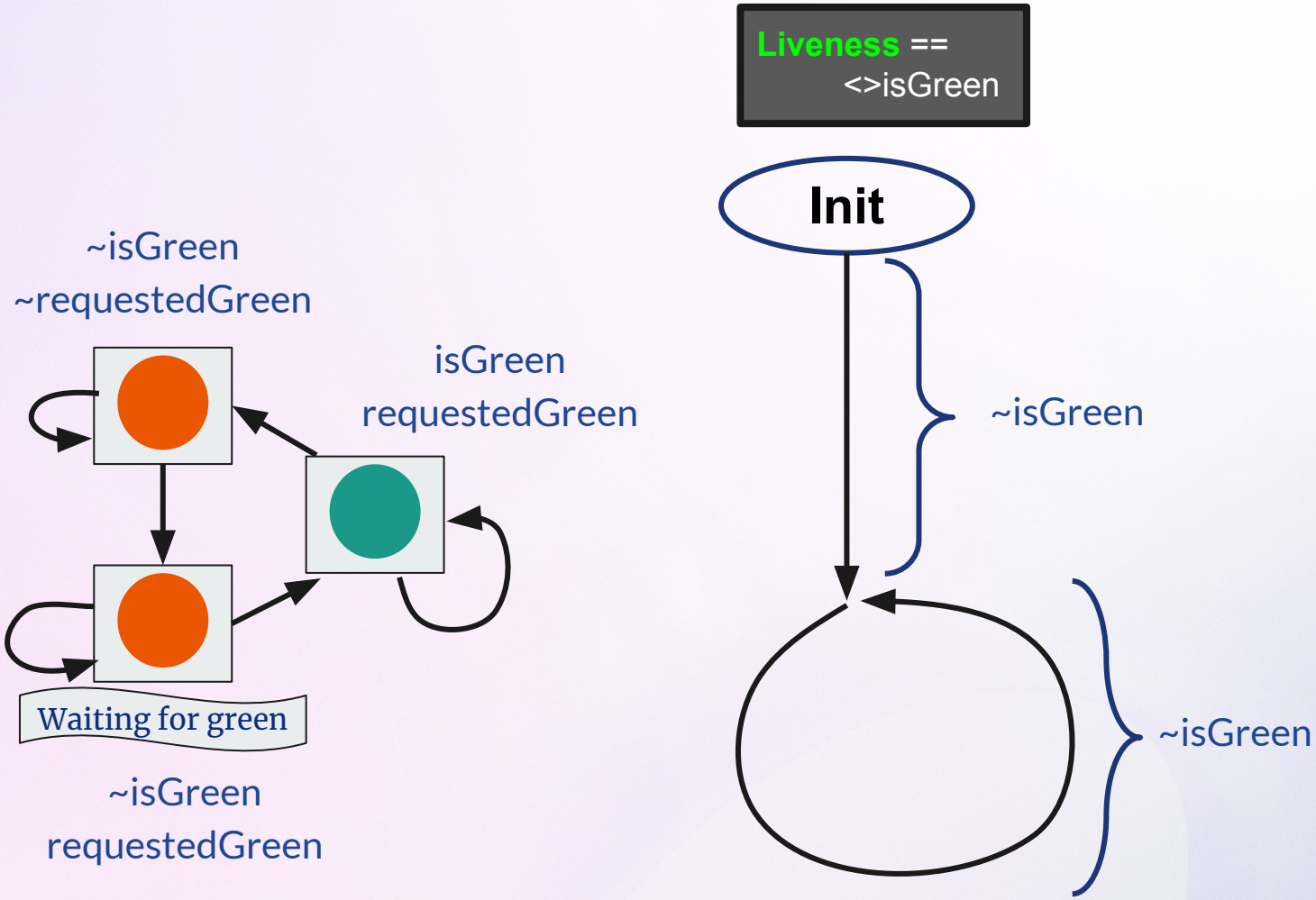




Not a counterexample!

Maybe isGreen could appear in the future?

Counterexamples to Liveness: Lassos



In *finite-state systems*:
Counterexamples to liveness are **lassos**

Next stop:
Encoding traces with lassos

Encoding Lassos

First loop state = Last loop state

=> Remember first loop state: Additional variables ("**History Variables**")

```
VARIABLES
        \* @type: Bool;
        InLoop,

        \* @type: Bool;
        loop_isGreen,

        \* @type: Bool;
        loop_requestedGreen
```

How do we know when the loop starts?
Nondeterministic **guessing**!

```
Next == …
/\ InLoop' \in BOOLEAN
/\ (InLoop => InLoop')
/\ (IF InLoop = InLoop'
        THEN UNCHANGED (<<loop_isGreen, loop_requestedGreen>>)
        ELSE loop_isGreen' = isGreen /\ loop_requestedGreen' = requestedGreen)
```

How do we know what's a valid **last state** for the loop?
Ensure **current state** is equal to the remembered **first state**!

```
LoopOK ==
/\ InLoop
/\ loop_isGreen = isGreen
/\ loop_requestedGreen = requestedGreen
```

Finding Loops that are Counterexamples

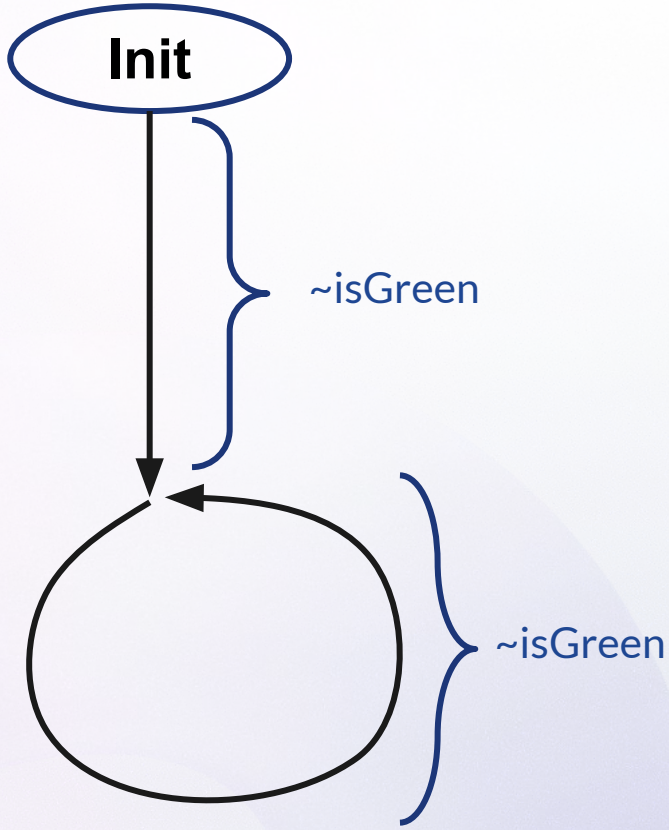
Finding traces that have a loop: ✔

Next step: Restrict to loops that are counterexamples

```
Liveness ==
        <>isGreen
```

Additional variable: **satisfied_(<>isGreen)** is true if and only if **isGreen** is true at some later point in the trace

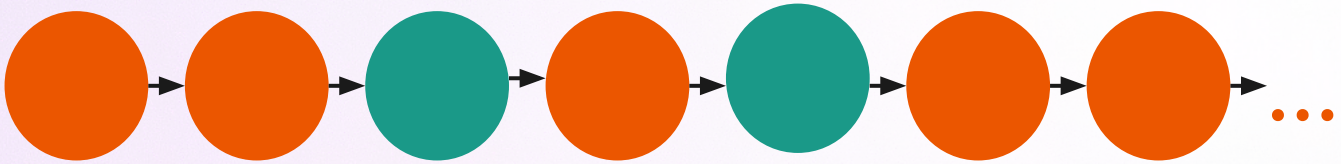
```
VARIABLE
        \* @type: Bool;
        satisfied_(<>isGreen)
```



satisfied_(<>isGreen) promises future behaviour!
("Prophecy variable")
Promises traces that satisfy/don't satisfy <>isGreen:

- Guess the value initially
- Only allow traces that match the guess

Prophecy Variables



satisfied_(<>isGreen)?

✔ ✔ ✔ ✔ ✔ ✘ ✘

satisfied_(<>isGreen) behaves as if it knew the future of the run!

Easy in TLA+:

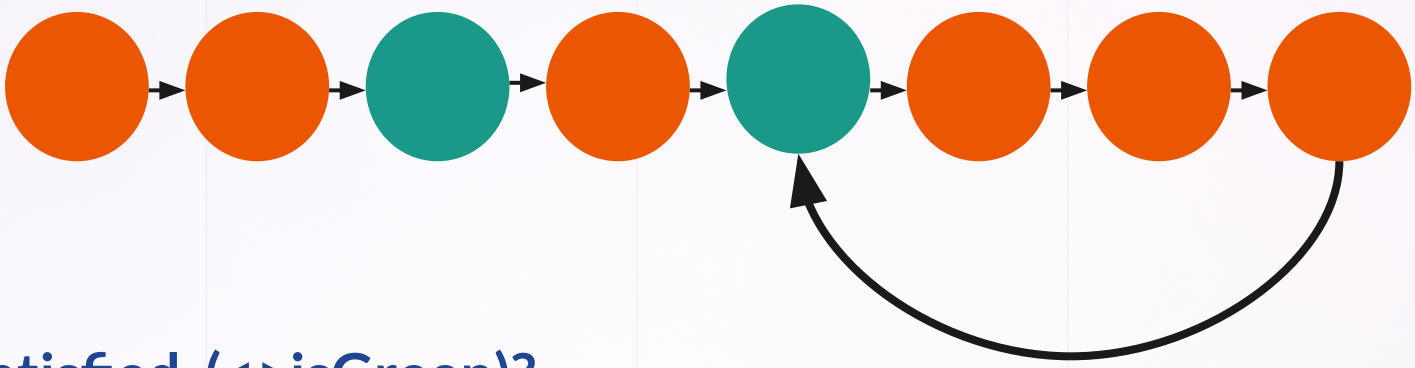
```
Next == …
/\ satisfied_(<>isGreen)' \in BOOLEAN
/\ satisfied_(<>isGreen) <=> isGreen \/ satisfied_(<>isGreen)'
```

…a bit harder for Apalache: Could introduce double-priming, which is not allowed!

Solution: Another promise variable that promises the next value of **satisfied_(<>isGreen)**

```
VARIABLE
    \* @type: Bool;
    satisfied_(<>isGreen)_next
```

Prophecy Variables & Loops



satisfied_(<>isGreen)?

✔ ✔ ✔ ✔ ✔ ✔ ✔ ✔

Promise variables in the last state depend on the promise variables in the first state of the loop
("roll over" to the start of the loop)

=> Save the status of promise variables when the loop starts

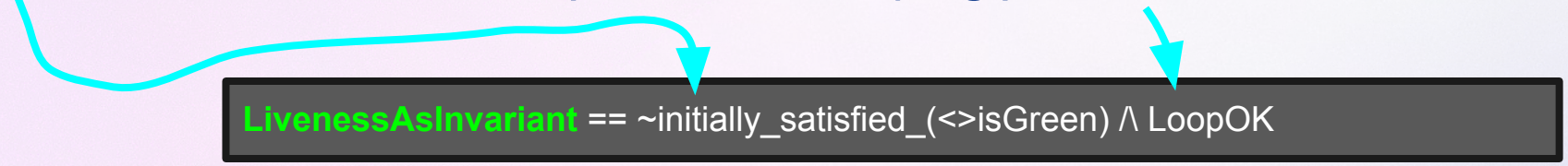
```
VARIABLE
        \* @type: Bool;
        loop_satisfied_(<>isGreen)
```

Encoding Temporal Properties

Prophecy Variables + Encoding Loops are enough to encode temporal properties

A trace is bad if:

- **~satisfied_(<>isGreen)** in the initial state, and
- we can close a loop (while satisfying promise variables)

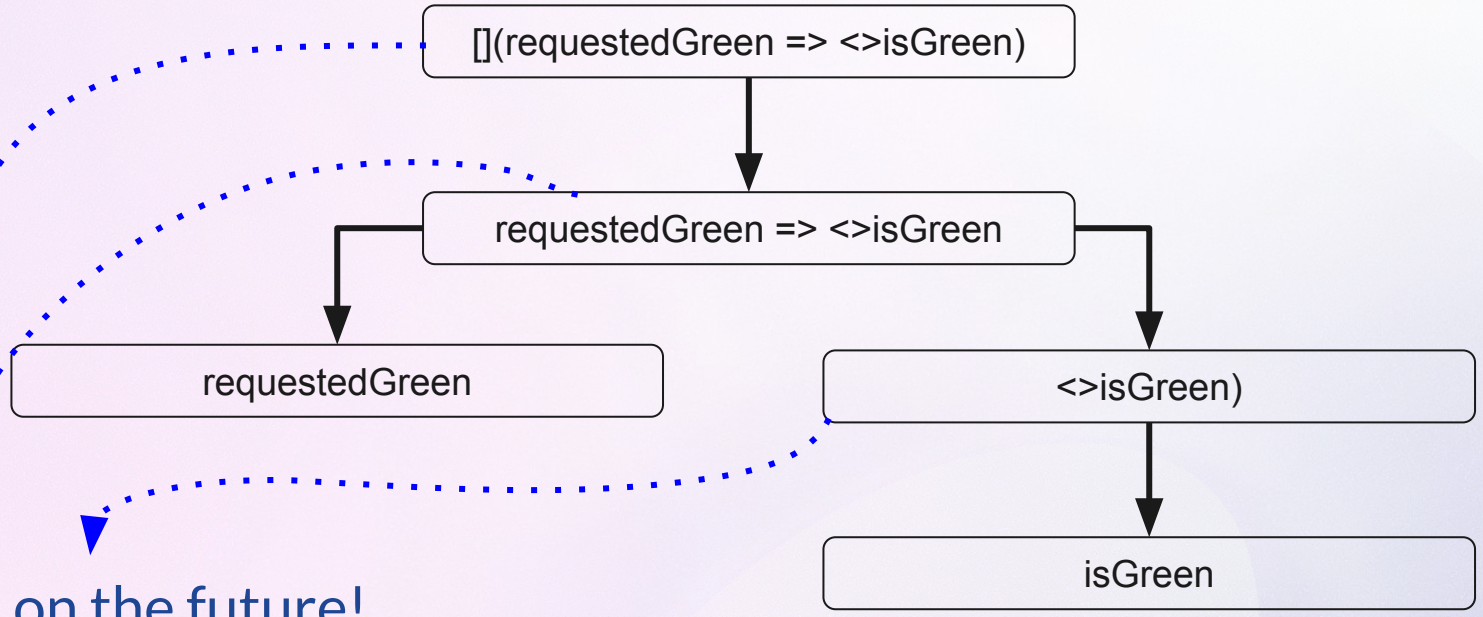


Thanks to promise and history variables, only depends on the current state, but still reasons about the whole trace!

Nested Temporal Properties

What about more complex temporal properties?

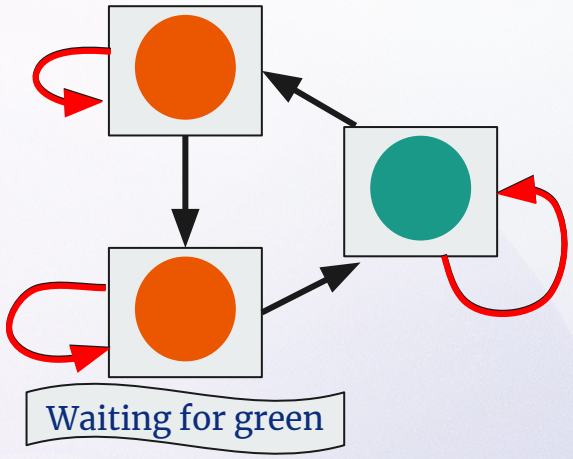
```
ComplexLiveness == [](requestedGreen => <>isGreen)
```



Depend on the future!

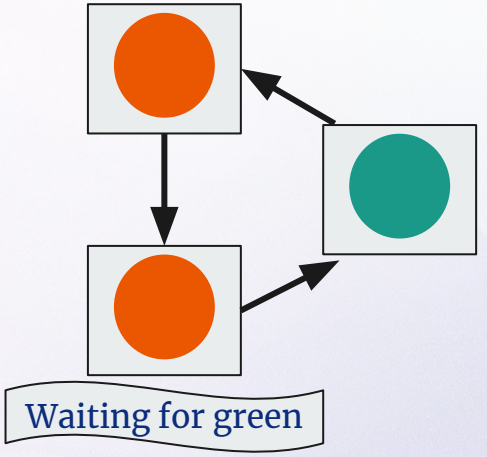
=> Prophecy variables for each!

TrafficLight in Action



Unstuttering TrafficLight

Liveness ==
<>isGreen



TrafficLight in Action

Encoding needs lots of extra variables — How many?

VARIABLES		
Original	**Liveness** == <>isGreen	**ComplexLiveness** == [](requestedGreen => <>isGreen)
2	10	16

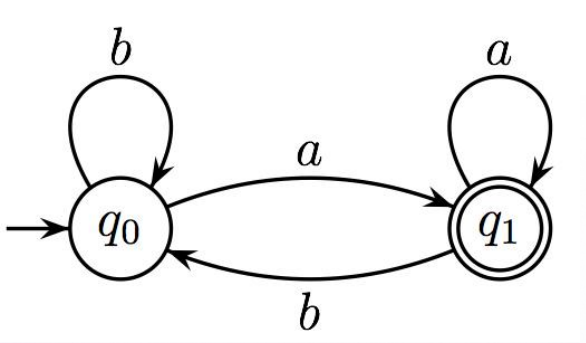
...but: extra variables cause almost no slowdown

Doubles the number of symbolic transitions (no matter the property!)

TRANSITIONS		
Original	**Liveness** == <>isGreen	**ComplexLiveness** == [](requestedGreen => <>isGreen)
4	8	8

Alternative Encodings

Temporal properties can be encoded as Büchi automata



Fewer variables

Automaton state = single integer

Visualization

Large or nondeterministic

Many extra symbolic transitions

Major slowdown for Apalache

Hard to understand at a glance

Alternative Encodings

Translate temporal properties
to trace invariants

```
TraceInvariant(hist) ==
        hist[Len(hist)].tokens =
                hist[1].tokens * 2
```

Straightforward translation

No extra variables

Can be slower

Traces can be
difficult to understand

Why was the property violated?

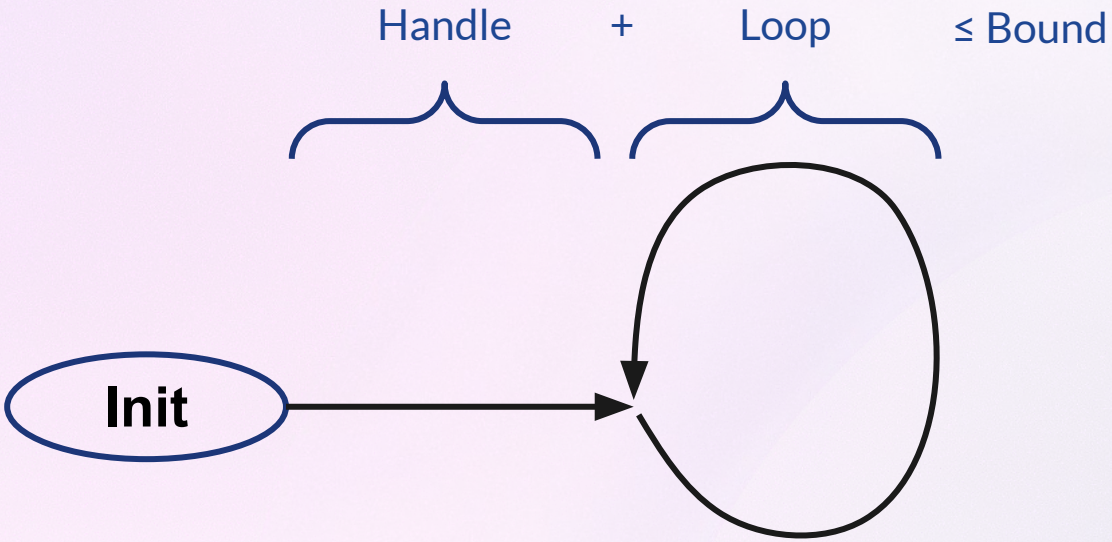
Bounded Model Checking & Liveness

Apalache: Symbolic bounded model checker
Reasons about traces of **finite** length

"There is no invariant violation in the first 50 steps"

What does this mean for lassos?
Bound on the length of the lasso: Handle + loop

"There is no counterexample lasso of size at most 50"



Fairness

Important for many temporal properties

Just need to handle ENABLED: Fairness can be rewritten

```
WF_vars(Next) <=>
        <>[](ENABLED <<A>>_v) => []<><<A>>_v
```

```
FairLiveness ==
        WF_vars(Next) => <> isGreen
```

...but: Apalache does not support Fairness and ENABLED

- No problem for safety, so no issue previously

Could address Fairness by adjusting Apalache internals, but:

- Hard to change
- Expensive to maintain

Instead: preprocess ENABLED

- Resilient to changes to internal transition execution
- ENABLED can be used outside of Fairness

Goal: Handling fairness via preprocessing!

Preprocessing ENABLED

Apalache has a symbolic transition finder – can help handling ENABLED

```
Action ==
  /\ x >= 5
  /\ x' = x + 1
  /\ y = x' + 5
  /\ z' \in {x', y}
  /\ y' = y
```

Split into

Assignments

```
x' := x + 1

\E v \in {x', y}: z' := v

y' := y
```

Replace with true!

Conditions

```
x >= 5

y = x' + 5
```

Replace with assignment to x'

```
ENABLED(Action) ==
        /\ x >= 5
        /\ y = (x + 1) + 5
        /\ \E v \in {(x + 1)  y}: TRUE
```

Straightforward rewriting!

But: restricted to expressions that are handled by the transition finder

```
ComplexAction ==
        y' * y' +1 = 0
```

Apalache now supports arbitrary temporal properties

Temporal properties are transformed into invariants using history and prophecy variables

Thanks for listening!

apalache.informal.systems
informal.systems
p-offtermatt.github.io
philip.offtermatt@informal.systems