GameSS

EDUCATIONAL MATERIAL IN CYBER SECURITY

Who is the material for?

This module provides a high-level introduction to Formal Software Verification and its application using TLA+, an industry-level model checker.

This course is suitable for software developers, software architects, technology/innovation officers with development experience, and students with some programming experience.

The practical parts of this module assume a working understanding of sets and first order logic.

Who made this material?

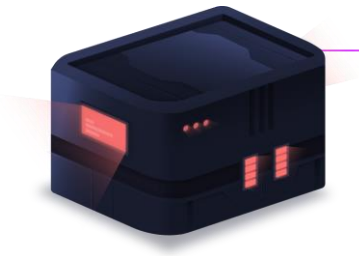
Marco Peressotti

Associate Professor at the Department of Mathematics and Computer Science at the University of Southern Denmark (SDU)

https://marcoperessotti.com

What are the main takeaways for this content?

- An overview of formal methods and their use for verifying software
- An introduction to software verification using TLA+

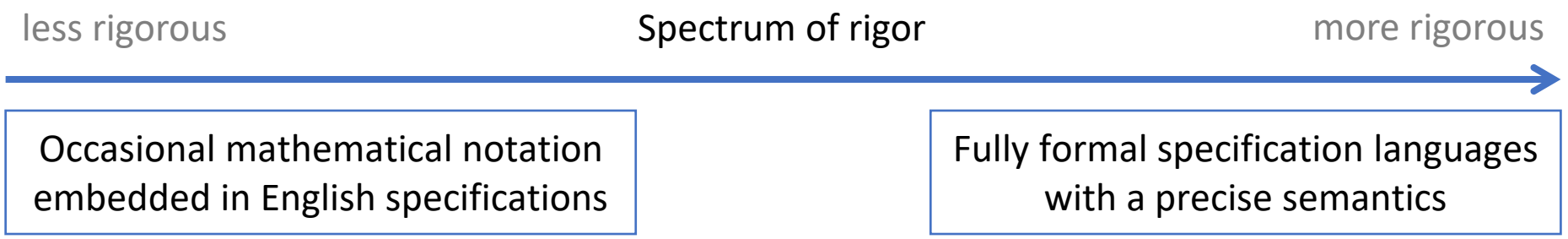


Formal Software Verification

A short introduction to Formal Methods

Formal Methods is a HUGE umbrella term

- In general, it refers to the use of mathematically rigorous techniques in software and hardware engineering
- Can assume various forms and levels of rigour

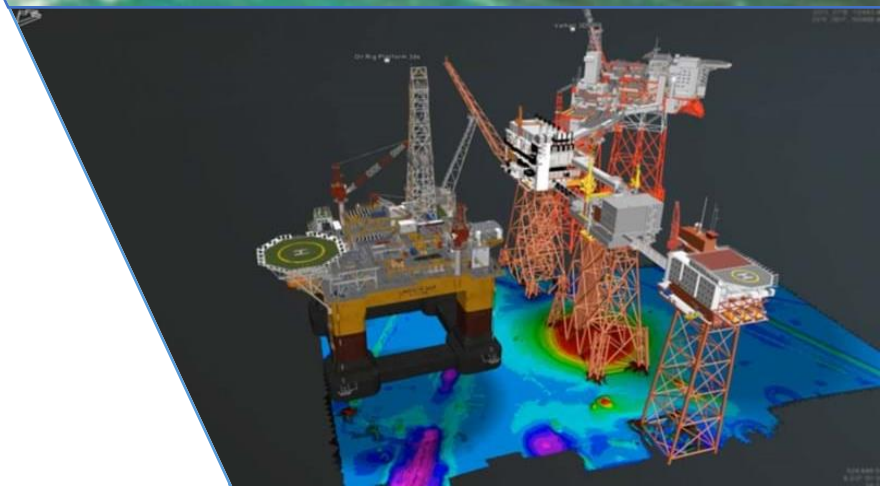


- Broad variety of fundamentals of theoretical computer science:

  Logics, Formal Languages, Automata Theory, Program Semantics, Type Theory, Behavioural Theory, Model Checking, …

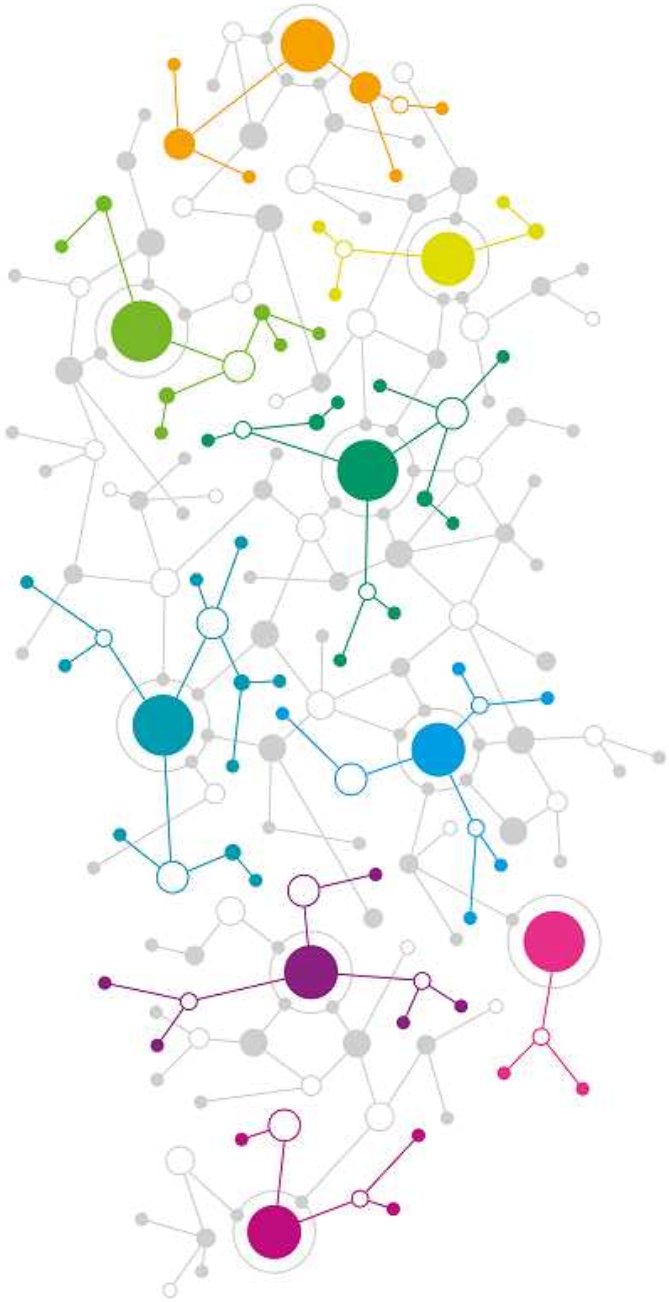
Analogy with other engineering disciplines

- Engineers in traditional disciplines build mathematical models of their designs
  - Use calculations to establish that the design, in the context of a modelled environment, satisfies its requirements
  - Iterate through phases: model, compute, interpret, repeat
- Computational solutions (e.g., CFD, FEM)
- Accuracy and faithfulness of the model must be verifiable (lab tests, prototypes,…)
- Requirements in certifications

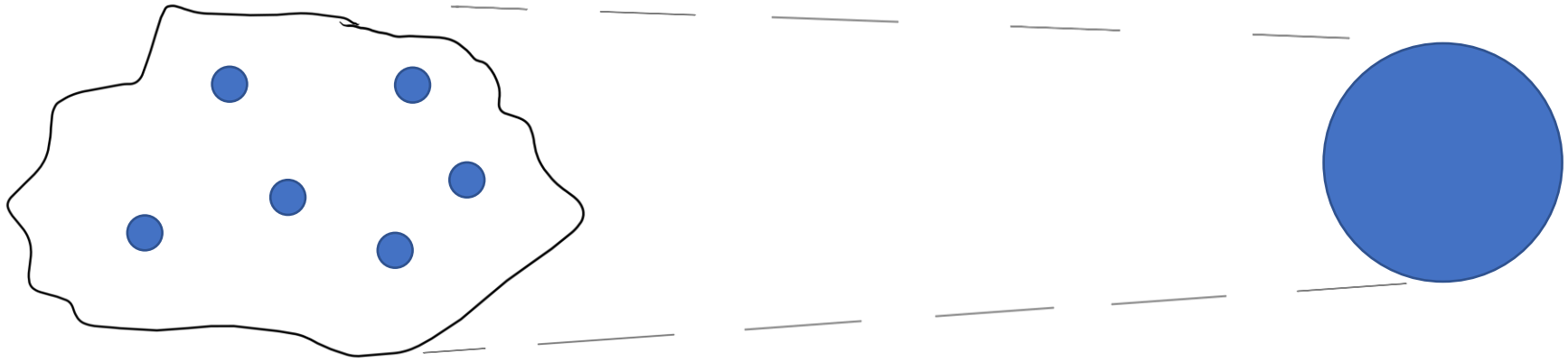


Analogy with other engineering disciplines

- Build mathematical models of computational systems
- Models are descriptions in some logical system
  - Unambiguous representation and meaning
  - Support rigorous and reproducible reasoning
- Mechanisation by automated reasoning
  - Automated theorem proving, model checking, static analysis, etc.
- Automated reasoning covers all modelled behaviours
  - Accurate models are used for verification (showing that the system has a property) and synthesis (generation or repair of code)
  - Approximate models are used for refutation (showing that the system does not have a property)



Testing & Simulation vs. Formal Methods



Done on real (or dev) systems

- Achieve only partial coverage of the system
- Provide results close to reality

Done on a model of the system

- Achieve complete coverage of the model
- Approximate reality

Testing & Simulation vs. Formal Methods

Testing shows the presence of errors, not their absence

Edsger Dijkstra

The quality of testing depends on finding the right tests

- Can be improved by coverage measures, input fuzzing (semi)automated test generation
- However, achieving good coverage is almost impossible
- Computational cost may impose trade-offs on coverage

So… it depends on skill and luck

FM show the absence of errors the model is accurate enough to manifest

The quality of models depends on finding the right level of abstraction for the model

- Some aspects of the real system might be overlooked
- Computational cost may impose trade-offs on the accuracy of the model

So… it depends on skill and luck

- Pentium FDIV Bug
  - A hardware bug in the early Pentium processor
  - In 1994 Intel recalled the defective processors losing ~500.000.000$

- Intel uses FM to verify its designs before production, still…there are bugs
  - Spectre and Meltdown exploit time-based side channels
  - Finding time-based attacks requires modelling time too

FM show the absence of errors the model is accurate enough to manifest

Taxonomy

- Level 0: Formal Specification
  - **Description of the system under development**
  - Unambiguous syntax, semantics and support rigorous (automated) reasoning
  - Development is often carried out informally or in combination with level 1 FM
  - One of the most **accessible and cost-effective** options in many cases
  - Tools: **TLA+**, CAAL, LOTOS, B, Z, VDM, …
- Level 1: Formal Verification and Development
  - Static analysis of systems, synthesis of programs, extraction of proof obligations
  - High-integrity systems or mission critical components of larger systems
  - Techniques: abstract interpretation, model checking, type systems, …
- Level 2: Theorem Provers
  - Fully formal machine-checked proofs of correctness
  - Tools: Coq, Agda, Isabelle, HOL, SMTs, …

When is it worth using FM?

The traditional answer

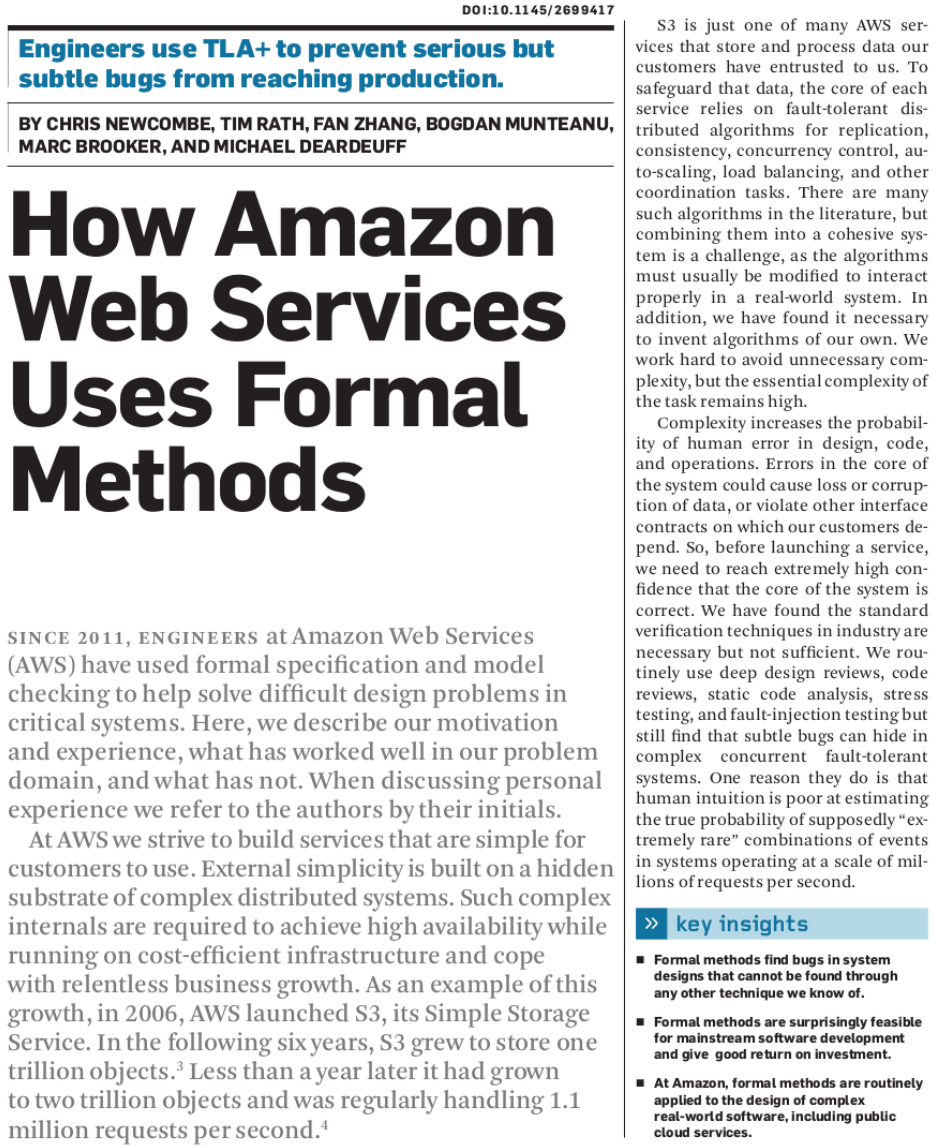
- Light-weight formal methods (e.g., formal specifications)
- Partial formalisation, focus on critical components (e.g., communication libraries)
- More mature tools many with industry backing (e.g., Infer, TLA+, UPAAL, etc.)
- Integration with development environments and DevOps/DevSecOps
- Education (MSc, BSc, Academies, etc.)

Adoption is on the rise thanks to

- High-integrity systems
  - Safety critical systems (aerospace, automotive, energy, etc)
  - Security critical systems (banking, voting, etc)
  - Mandated by industry standards (IEC61508, DO-178B)
- Systems where early error detection saves lots of money (e.g. hardware)
- Systems with unpredictable environments (e.g., concurrent and distributed systems such as Cloud, Edge, IoT)

FM adoption

- Amazon has used FM on 14 large complex systems.
- In each, FM has added significant value,
  - finding subtle bugs we would not have found by other means.
  - giving us enough confidence to make aggressive optimizations without sacrificing correctness.
- Amazon has 7 teams using FM
- Engineers at all levels learned FM from scratch and get useful results in 2-3 weeks.



DOI:10.1145/2699417

Engineers use TLA+ to prevent serious but subtle bugs from reaching production.

BY CHRIS NEWCOMBE, TIM RATH, FAN ZHANG, BOGDAN MUNTEANU, MARC BROOKER, AND MICHAEL DEARDEUFF

How Amazon Web Services Uses Formal Methods

SINCE 2011, ENGINEERS at Amazon Web Services (AWS) have used formal specification and model checking to help solve difficult design problems in critical systems. Here, we describe our motivation and experience, what has worked well in our problem domain, and what has not. When discussing personal experience we refer to the authors by their initials.

At AWS we strive to build services that are simple for customers to use. External simplicity is built on a hidden substrate of complex distributed systems. Such complex internals are required to achieve high availability while running on cost-efficient infrastructure and cope with relentless business growth. As an example of this growth, in 2006, AWS launched S3, its Simple Storage Service. In the following six years, S3 grew to store one trillion objects.[3] Less than a year later it had grown to two trillion objects and was regularly handling 1.1 million requests per second.[4]

S3 is just one of many AWS services that store and process data our customers have entrusted to us. To safeguard that data, the core of each service relies on fault-tolerant distributed algorithms for replication, consistency, concurrency control, auto-scaling, load balancing, and other coordination tasks. There are many such algorithms in the literature, but combining them into a cohesive system is a challenge, as the algorithms must usually be modified to interact properly in a real-world system. In addition, we have found it necessary to invent algorithms of our own. We work hard to avoid unnecessary complexity, but the essential complexity of the task remains high.

Complexity increases the probability of human error in design, code, and operations. Errors in the core of the system could cause loss or corruption of data, or violate other interface contracts on which our customers depend. So, before launching a service, we need to reach extremely high confidence that the core of the system is correct. We have found the standard verification techniques in industry are necessary but not sufficient. We routinely use deep design reviews, code reviews, static code analysis, stress testing, and fault-injection testing but still find that subtle bugs can hide in complex concurrent fault-tolerant systems. One reason they do is that human intuition is poor at estimating the true probability of supposedly "extremely rare" combinations of events in systems operating at a scale of millions of requests per second.

» key insights

- **Formal methods find bugs in system designs that cannot be found through any other technique we know of.**
- **Formal methods are surprisingly feasible for mainstream software development and give good return on investment.**
- **At Amazon, formal methods are routinely applied to the design of complex real-world software, including public cloud services.**

FM adoption

- Amc
  com
- In e
  valu
  - f
    h
  - g
    c
    s
- Amc
- Engi
  from
  in 2

Applying TLA+ to some of Amazon's more complex systems.

System	Components	Line Count (Excluding Comments)	Benefit
S3	Fault-tolerant, low-level network algorithm	804 PlusCal	Found two bugs, then others in proposed optimizations
	Background redistribution of data	645 PlusCal	Found one bug, then another in the first proposed fix
DynamoDB	Replication and group-membership system	939 TLA+	Found three bugs requiring traces of up to 35 steps
EBS	Volume management	102 PlusCal	Found three bugs
Internal distributed lock manager	Lock-free data structure	223 PlusCal	Improved confidence though failed to find a liveness bug, as liveness not checked
	Fault-tolerant replication-and-reconfiguration algorithm	318 TLA+	Found one bug and verified an aggressive optimization

many AWS ser-
process data our
usted to us. To
the core of each
ult-tolerant dis-
for replication,
ency control, au-
ncing, and other
There are many
ne literature, but
o a cohesive sys-
s the algorithms
lified to interact
orld system. In
und it necessary
of our own. We
nnecessary com-
tial complexity of
ses the probabil-
in design, code,
rs in the core of
se loss or corrup-
e other interface
ur customers de-
nching a service,
remely high con-
of the system is
nd the standard
es in industry are
fficient. We rou-
gn reviews, code
analysis, stress
ection testing but
bugs can hide in
t fault-tolerant
they do is that
oor at estimating
f supposedly "ex-
nations of events
at a scale of mil-
second.

s

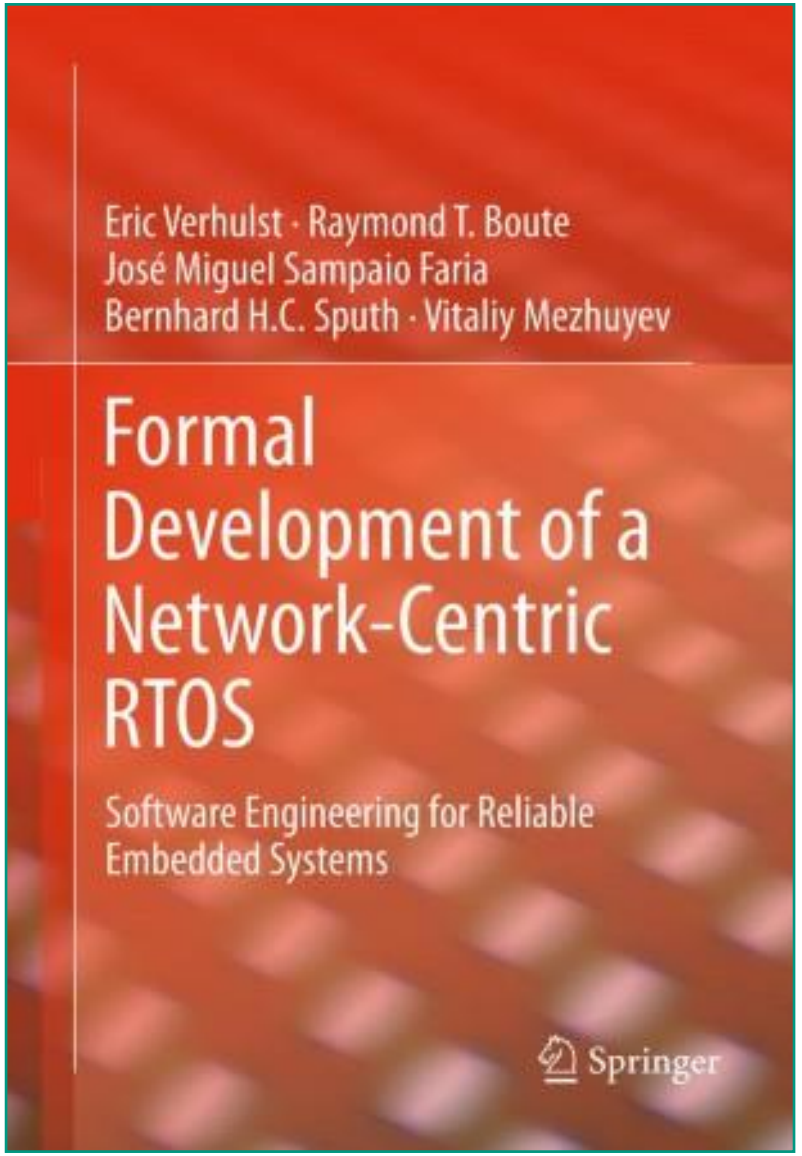
ugs in system
found through
e know of.

urprisingly feasible
are development
on investment.

thods are routinely
f complex
ncluding public

FM adoption

- A real-time operating system designed using FM.
- FM found subtle and severe security bugs.
- Developing an abstract model to apply FM had beneficial impacts beyond just finding bugs:
  - The level of abstraction helped in designing a much cleaner architecture.
  - The resulting codebase was 10 times smaller than the previous version of the OS despite the addition of new features
  - This level of reduction cannot be achieved with simple refactoring; it requires developing a deeper understanding of the system, its parts, and their interactions.

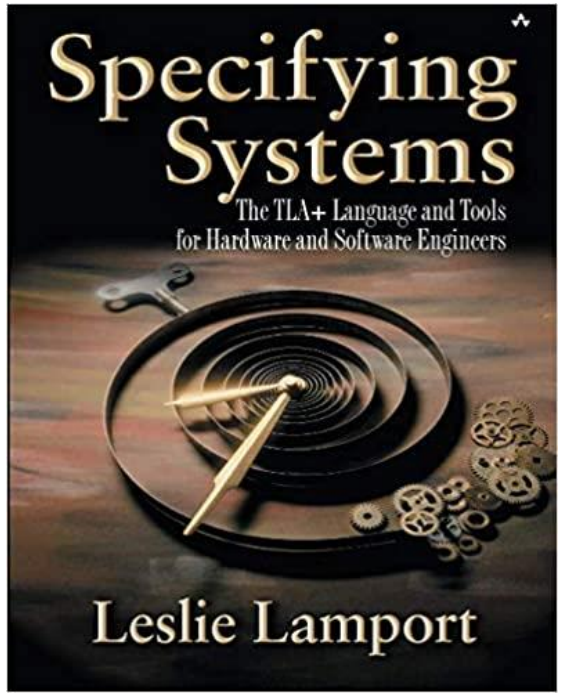


TLA+

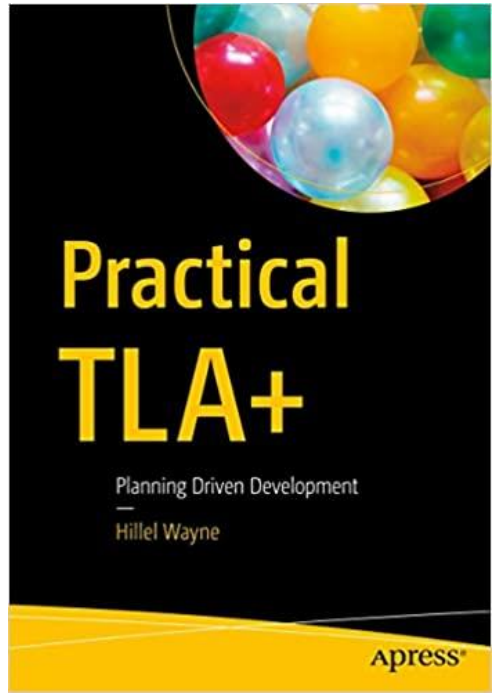
A toolbox for formal specifications

Resources

TLA+ Home Page
https://lamport.azurewebsites.net/tla/tla.html



Specifying Systems
Leslie Lamport



Practical TLA+
Hillel Wayne

GameSS TLA+ rxamples and solutions https://bit.ly/gamess-tla

TLA+

- A language for high-level modelling of digital systems.
  - Formal specification (Level 0) above code-level
    - Language independent
    - Encourages abstracting lower-level details and focus on critical parts of systems
  - Used from algorithms to programs and complex systems, sequential or concurrent

- Has tools for checking those models.
  - The most important is TLC, a finite **model checker**
  - Especially suitable for concurrent systems (which are hard to test)
  - Catches errors before code is written

Model checking

- Systems are modelled as state machines

  State Machines are abstract machines that can evolve in discrete steps (called state transitions or just transitions)

- Properties are expressed as formulae in temporal logics

  A language for writing properties of that depend on time (e.g., the user is authenticated before modifying the document) and a set of formal rules for reasoning about these properties.

- Model Checkers are tools that automatically verify whether a state machine satisfies a formula
  - Popular in industry thanks to the low learning curve (compared to other approaches)
  - Tools: **TLA+**, UPAAL, PRISM, Alloy, …

System

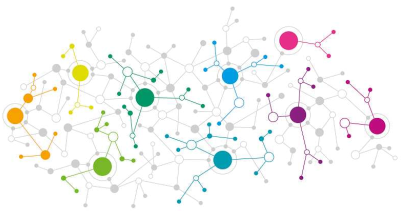
enjoys

Properties

State Machine (Model)

$\models$

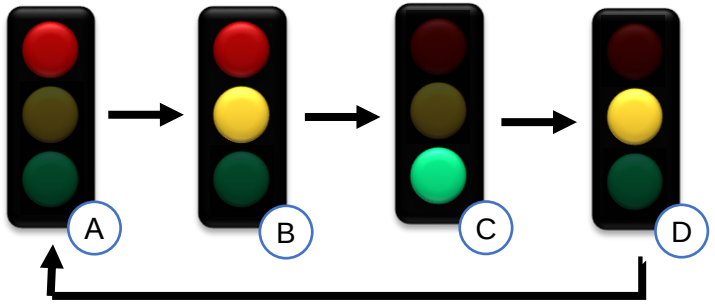
Temporal logic formulae



$$(\forall i : C(\phi_i) \vee (\exists j < i : B(\phi_j)))$$

Example: a traffic light controller

Desired behaviour



Implementation (v1)

State (variables)

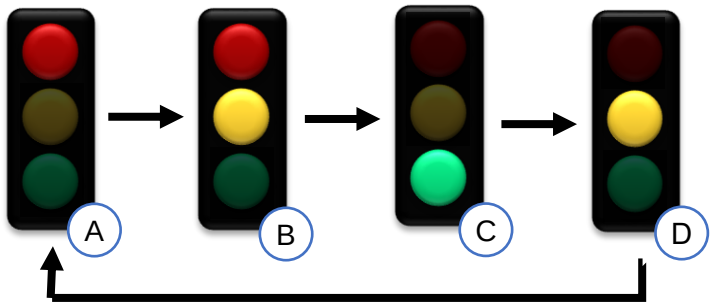
- Lights (on/off)
- Timer

Dynamics (actions)

1. If wait then
2. If wait then
3. If wait then
4. If wait then

Example: a traffic light controller

Desired behaviour



Implementation (v1)

State (variables)

- Lights (on/off)
- Timer

Dynamics (actions)

1. If wait then
2. If wait then
3. If wait then
4. If wait then

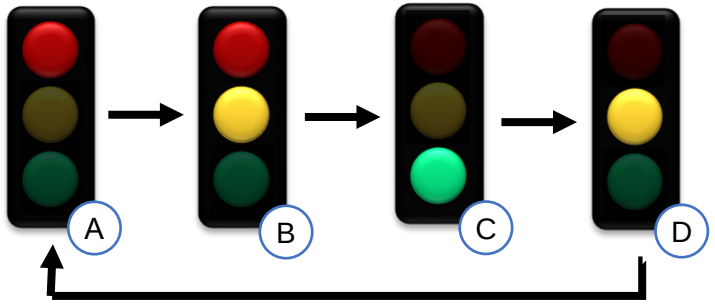
Model

State (variables)

- Lights (on/off)

Example: a traffic light controller

Desired behaviour



Implementation (v1)

State (variables)

- Lights (on/off)
- Timer

Dynamics (actions)

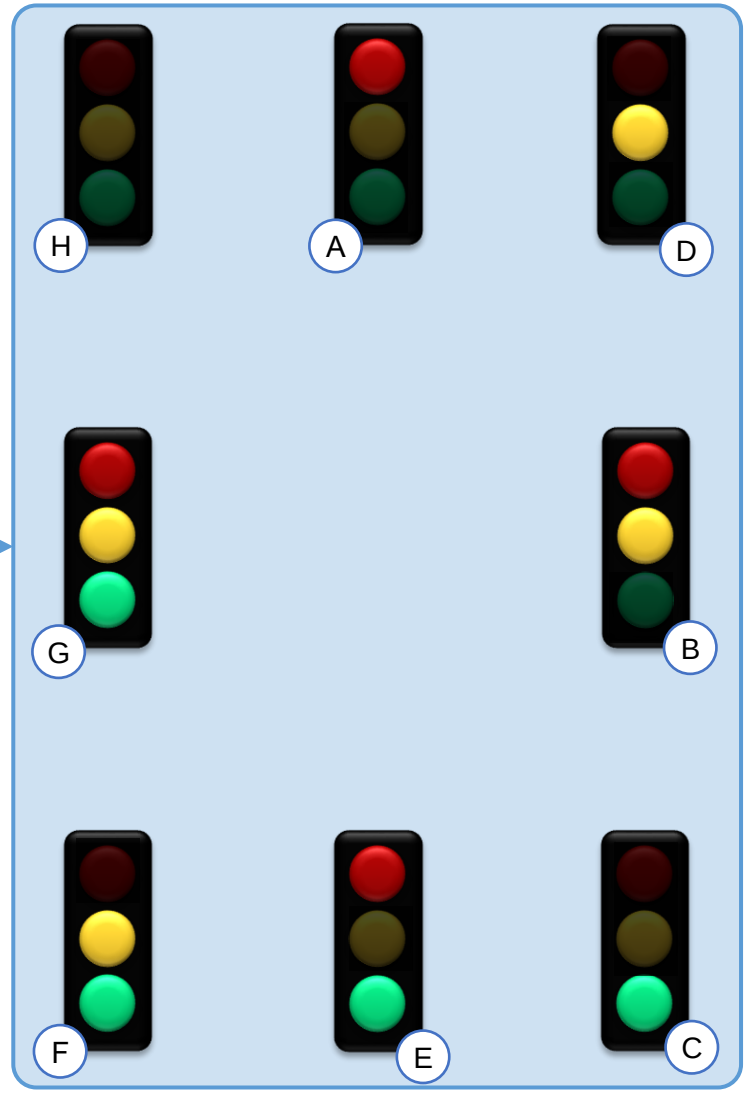
1. If wait then
2. If wait then
3. If wait then
4. If wait then

Model

State (variables)

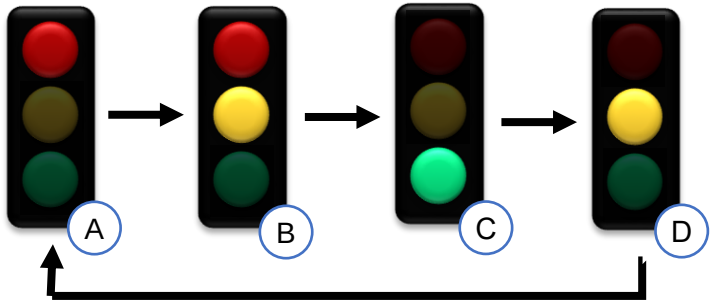
- Lights (on/off)

The values that these variables can take, define 8 states (the model does not include time)



Example: a traffic light controller

Desired behaviour



Implementation (v1)

State (variables)

- Lights (on/off)
- Timer

Dynamics (actions)

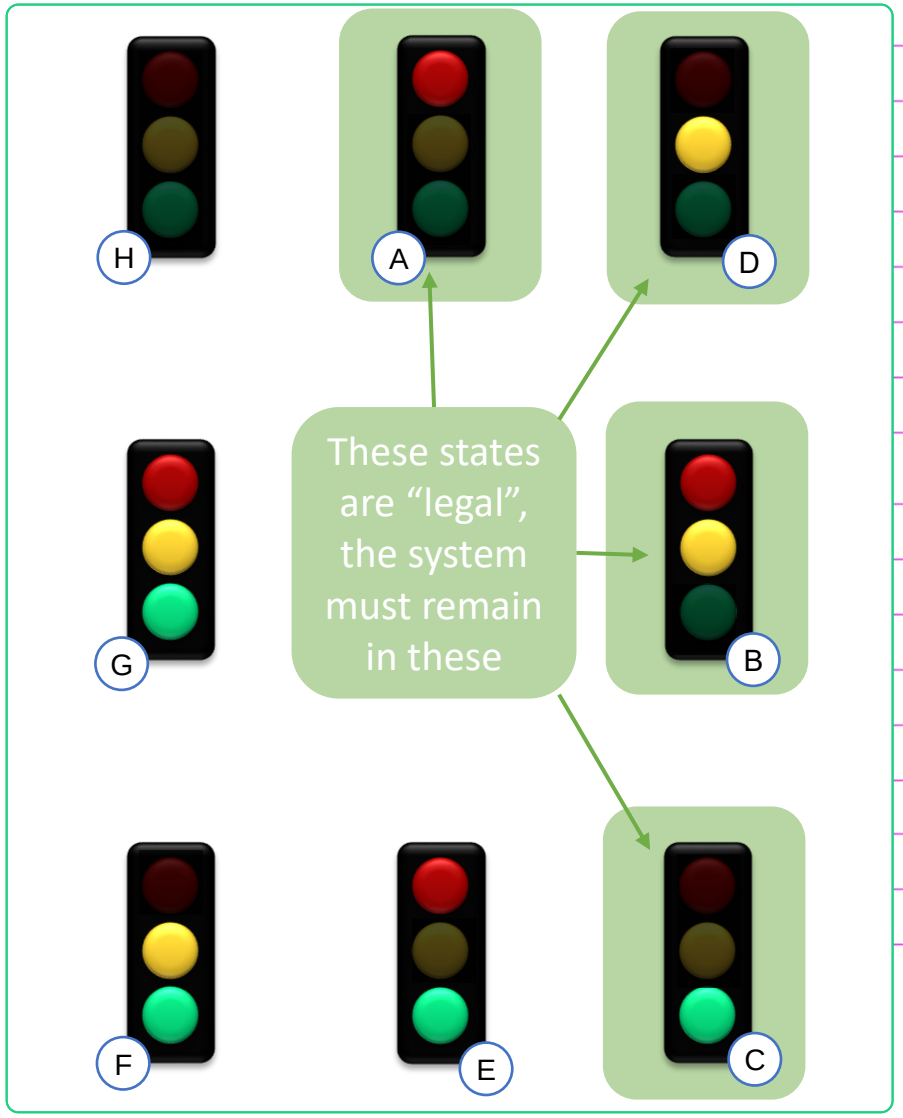
1. If wait then
2. If wait then
3. If wait then
4. If wait then

Model

State (variables)

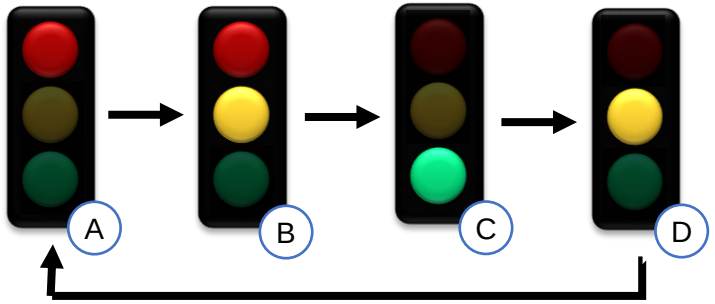
- Lights (on/off)

The values that these variables can take, define 8 states (the model does not include time)



Example: a traffic light controller

Desired behaviour



Implementation (v1)

State (variables)

- Lights (on/off)
- Timer

Dynamics (actions)

1. If wait then
2. If wait then
3. If wait then
4. If wait then

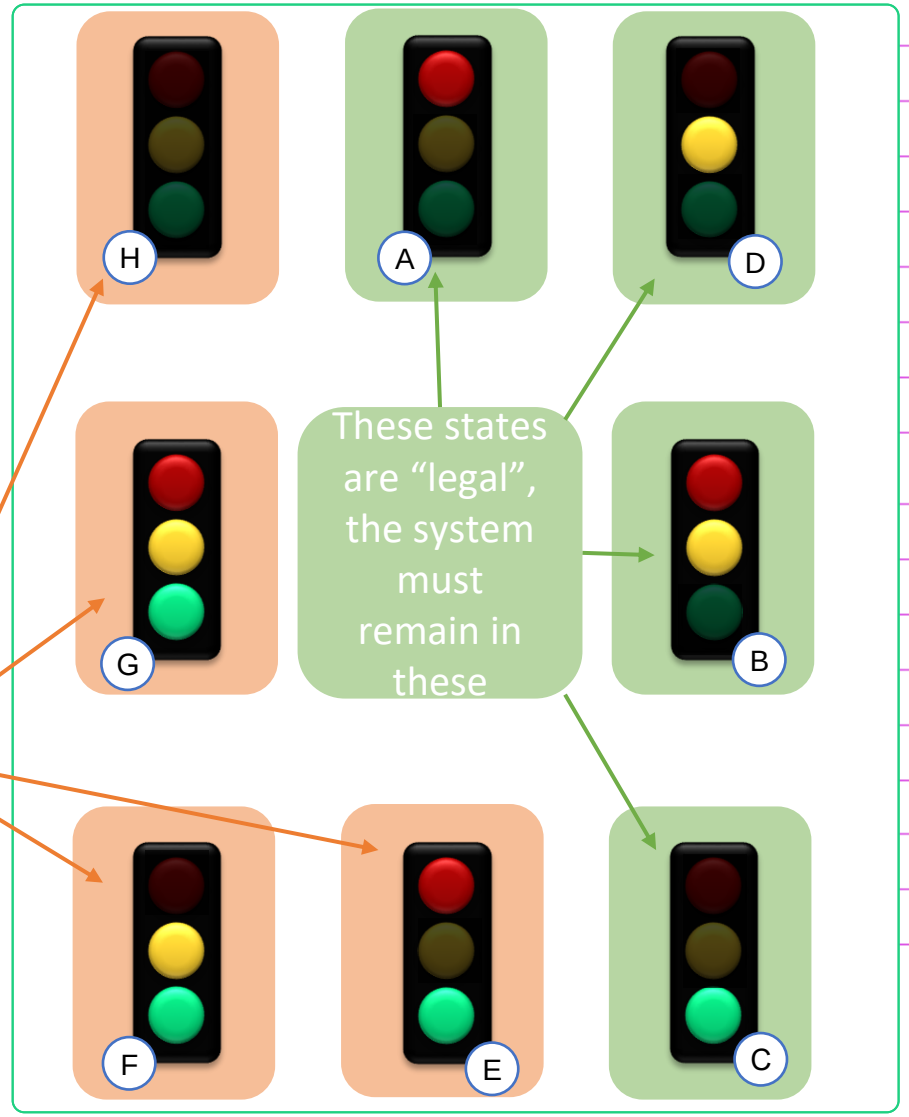
Model

State (variables)

- Lights (on/off)

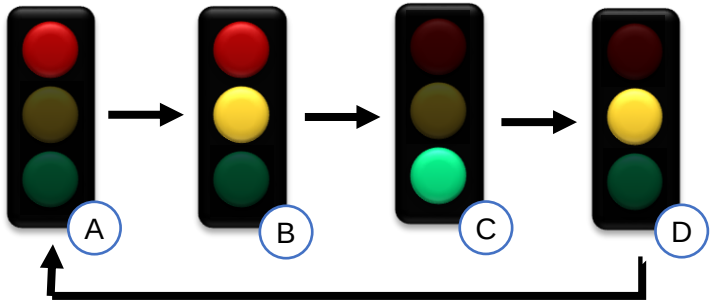
The values that these variables can take, define 8 states (the model does not include time)

These states are “illegal”: the system must never reach any of them



Example: a traffic light controller

Desired behaviour



Implementation (v1)

State (variables)

- Lights (on/off)
- Timer

Dynamics (actions)

1. If wait then
2. If wait then
3. If wait then
4. If wait then

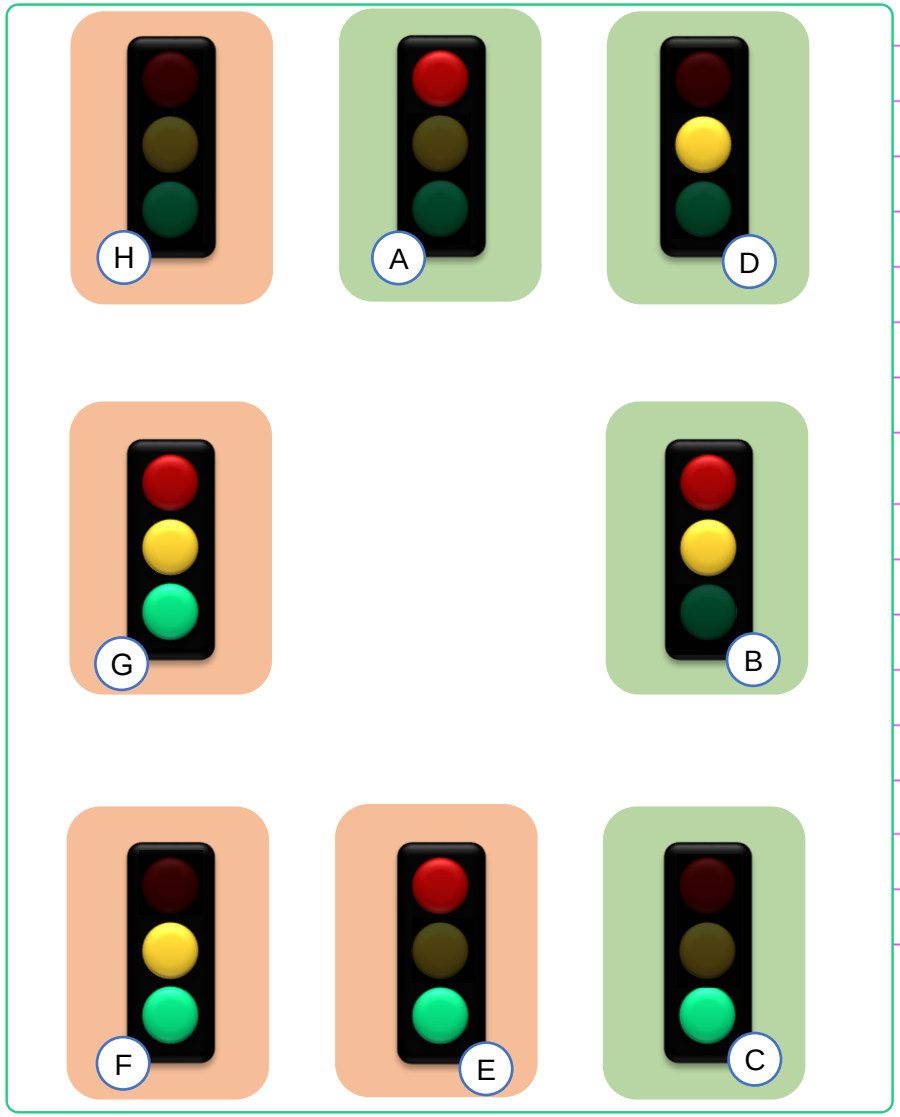
Model

State (variables)

- Lights (on/off)

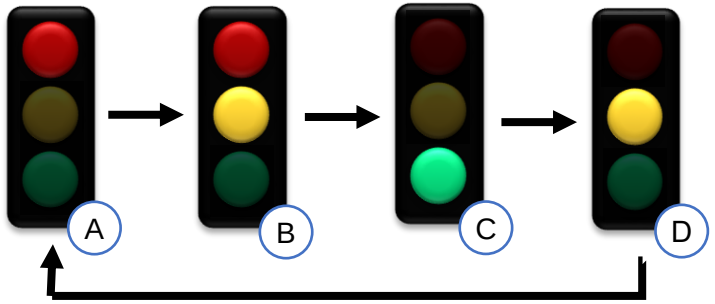
Dynamics (actions)

1. If then
2. If then
3. If then
4. If then



Example: a traffic light controller

Desired behaviour



Implementation (v1)

State (variables)

- Lights (on/off)
- Timer

Dynamics (actions)

1. If wait then
2. If wait then
3. If wait then
4. If wait then

Model

State (variables)

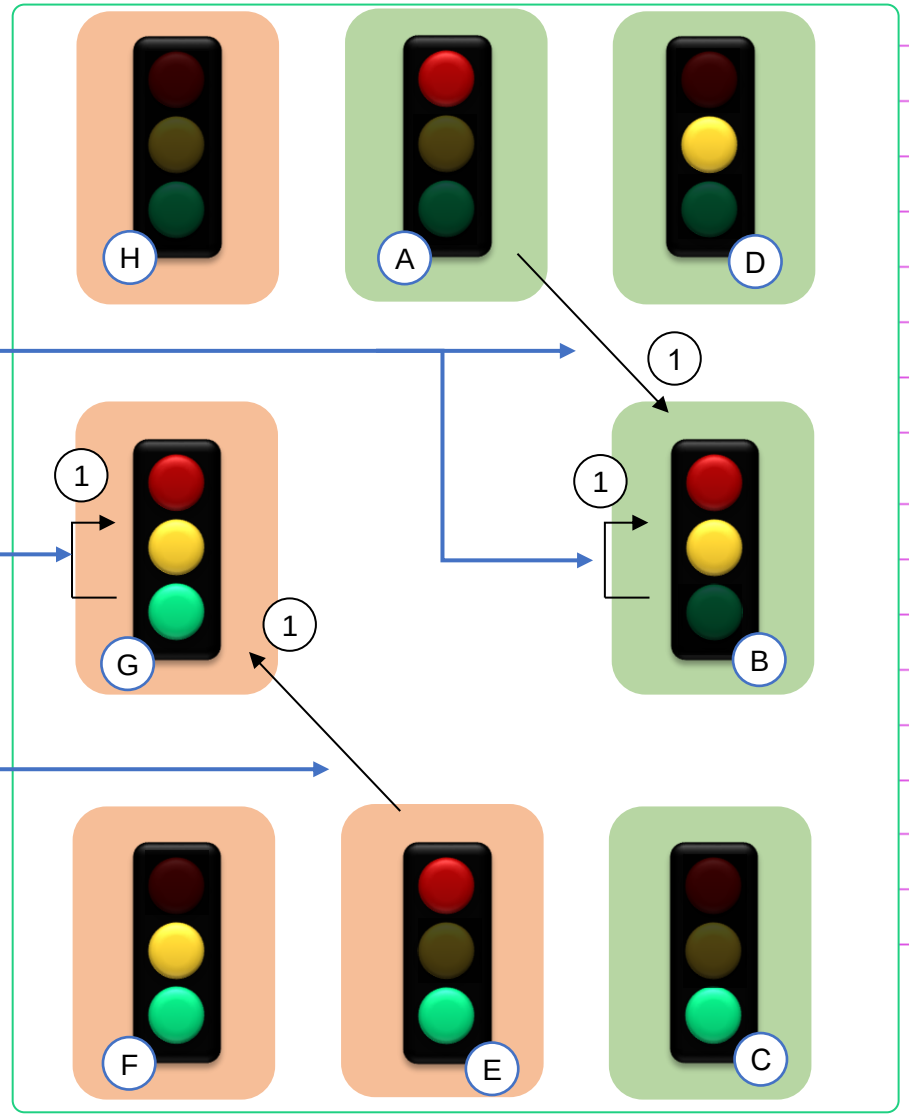
- Lights (on/off)

Dynamics (actions)

1. If then
2. If then
3. If then
4. If then

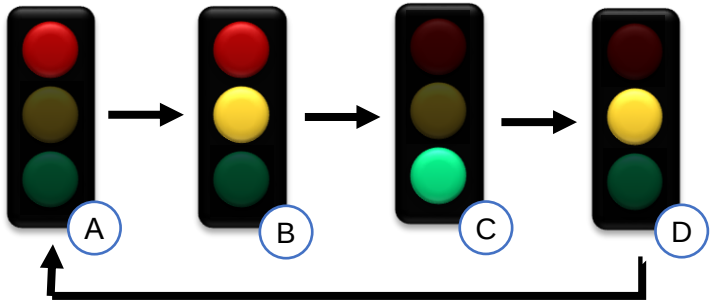
Rule 1: if the red light is on then, turn the yellow light on.

Rule 1: defines these state transitions



Example: a traffic light controller

Desired behaviour



Implementation (v1)

State (variables)

- Lights (on/off)
- Timer

Dynamics (actions)

1. If wait then
2. If wait then
3. If wait then
4. If wait then

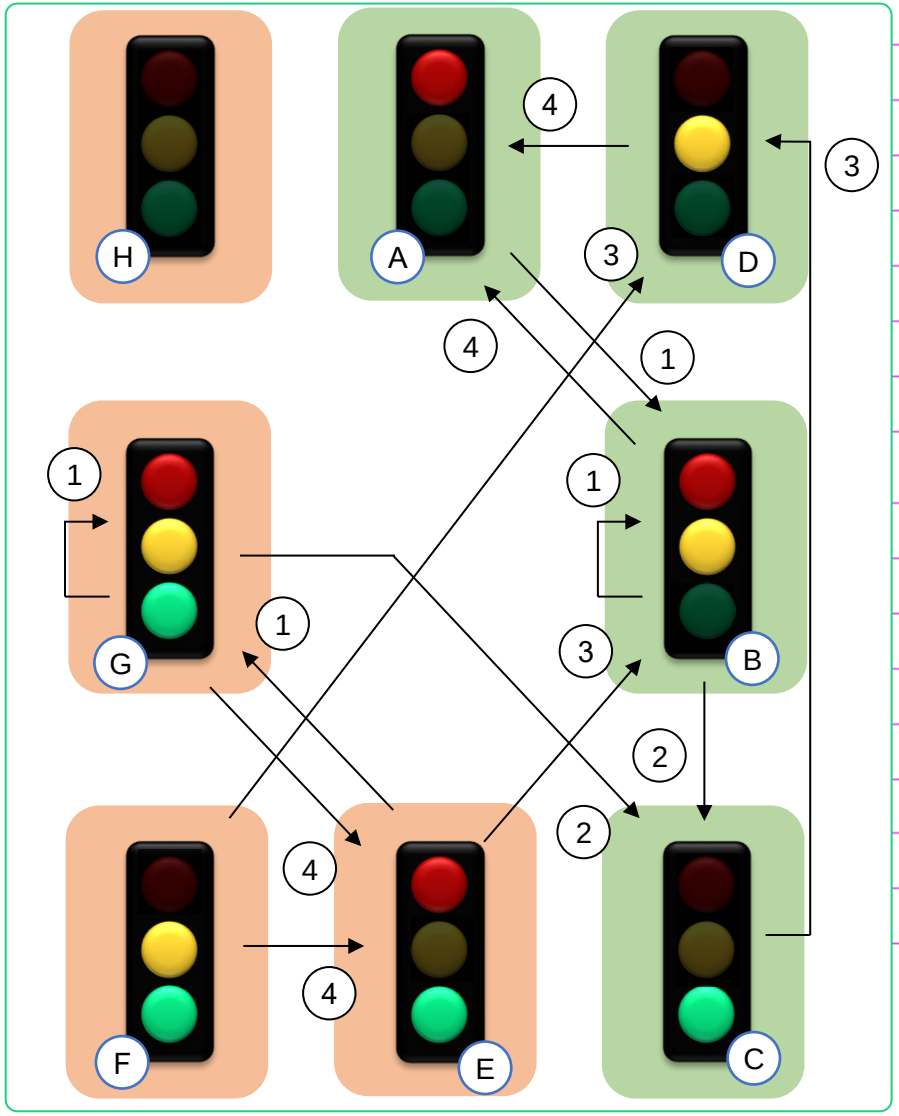
Model

State (variables)

- Lights (on/off)

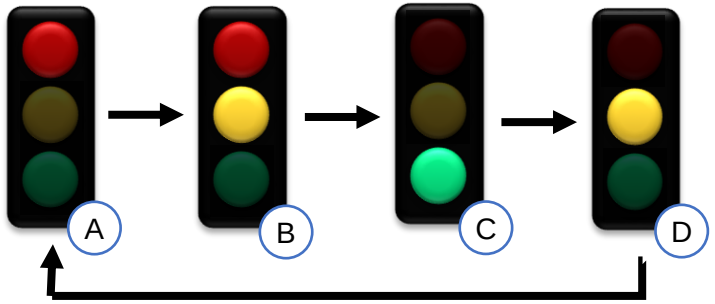
Dynamics (actions)

1. If then
2. If then
3. If then
4. If then



Example: a traffic light controller

Desired behaviour



Implementation (v1)

State (variables)

- Lights (on/off)
- Timer

Dynamics (actions)

1. If wait then
2. If wait then
3. If wait then
4. If wait then

Model

State (variables)

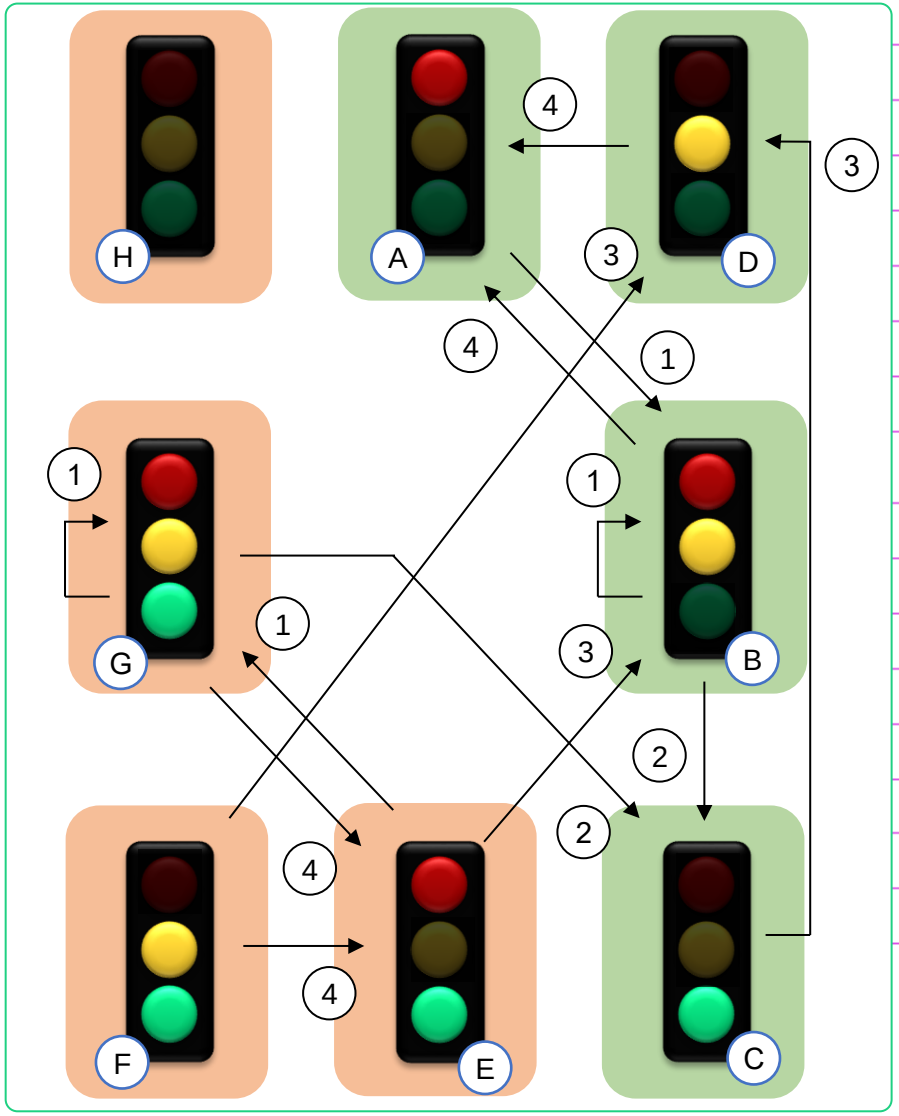
- Lights (on/off)

Dynamics (actions)

1. If then
2. If then
3. If then
4. If then

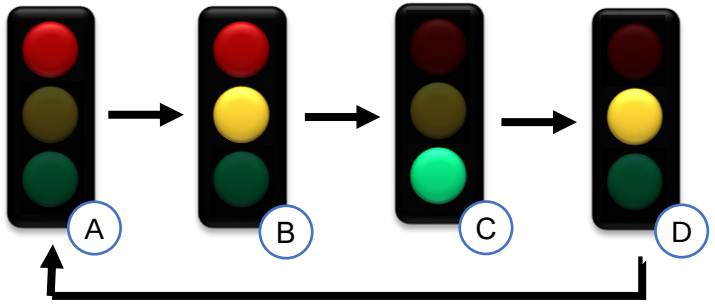
Properties

- Never stops
- Cannot reach an illegal state from a legal one
- after
- ….



Example: a traffic light controller

Desired behaviour



Implementation (v1)

State (variables)

- Lights (on/off)
- Timer

Dynamics (actions)

1. If wait then
2. If wait then
3. If wait then
4. If wait then

Model

State (variables)

- Lights (on/off)

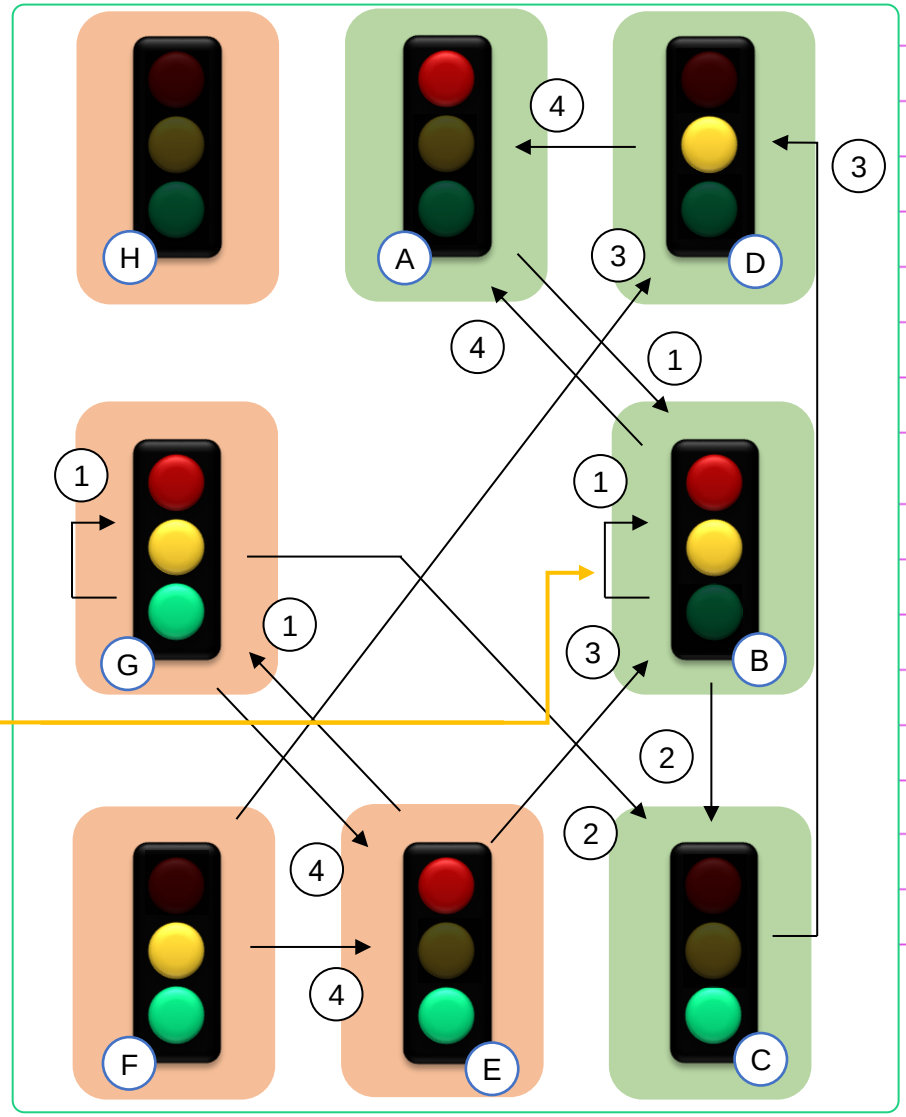
Dynamics (actions)

1. If then
2. If then
3. If then
4. If then

Properties

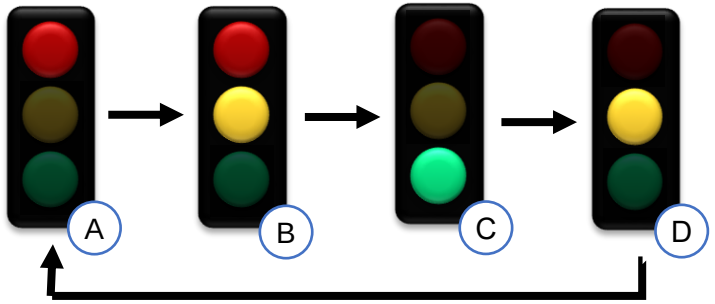
- Never stops
- Cannot reach an illegal state from a legal one
- after
- ….

Transitions violate the temporal property



Example: a traffic light controller

Desired behaviour



Implementation (v1)

State (variables)

- Lights (on/off)
- Timer

Dynamics (actions)

1. If wait then
2. If wait then
3. If wait then
4. If wait then

Model

State (variables)

- Lights (on/off)

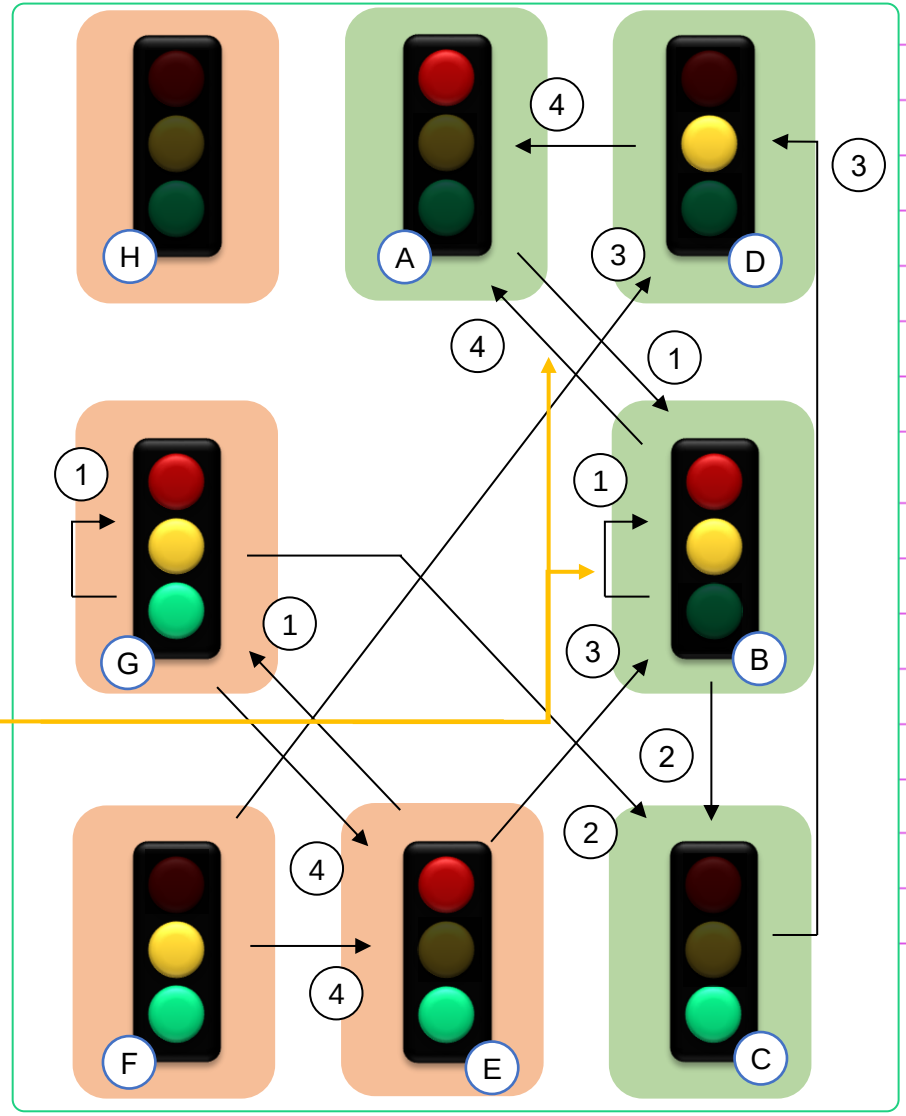
Dynamics (actions)

1. If then
2. If then
3. If then
4. If then

Properties

- Never stops
- Cannot reach an illegal state from a legal one
- after
- ….

Transitions violate the temporal property



Example: a traffic light controller in TLA+ (1)

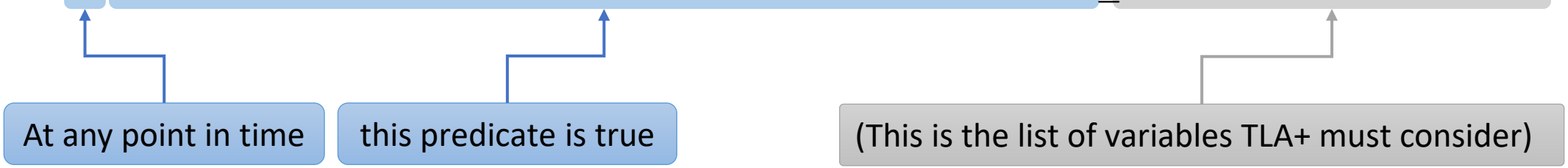
```
\* State
VARIABLES red, yellow, green

\* Invariants (predicates that must hold at any point in time)
\* TypesOk: variables are only On or Off
TypesOk == {red, yellow, green} \subseteq { On, Off }
\* LegalState: red or yellow are on if, and only if, green is off
LegalState == ( red = On  \/ yellow = On ) <=> ( green = Off )
```

Example: a traffic light controller in TLA+ (2)

```
\* Dynamics

\* Rule1: there is a transition from any state where red is on to any state
where yellow is on and red and green are the same (variables in the new state
are denoted with a prime).

Rule1 == ( red = On ) /\ ( yellow' = On /\ red’ = red /\ yellow’ = yellow )

Rule2 == ( yellow = On ) /\ ( green' = On /\ red' = Off /\ yellow' = Off )

\* `UNCHANGED vars` is a shorthand for `vars’ = vars`

Rule3 == ( green = On ) /\ ( green' = Off /\ yellow' = On /\ UNCHANGED red )

Rule4 == ( yellow = On ) /\ ( red' = On /\ yellow' = Off /\ UNCHANGED green )

\* Next: there is a transition between two states if any of the four rules can
be applied (by convention, `Next` is used to define all transitions of the
system)

Next == Rule1 \/ Rule2 \/ Rule3 \/ Rule4

\* Init: the system starts in the state with only red on (by convention,
`Init` is used to define the initial state or states)

Init == red = On /\ yellow = Off /\ green == Off
```

Example: a traffic light controller in TLA+ (3)

```
\* Temporal Properties

\* GreenAfterRedAndYellow: at any point in time, if red and yellow are on,
then in the next state green will be on

GreenAfterRedAndYellow ==

    [][( red = On  /\ yellow = On ) => (green' = On)]_<<red,yellow,green>>
```



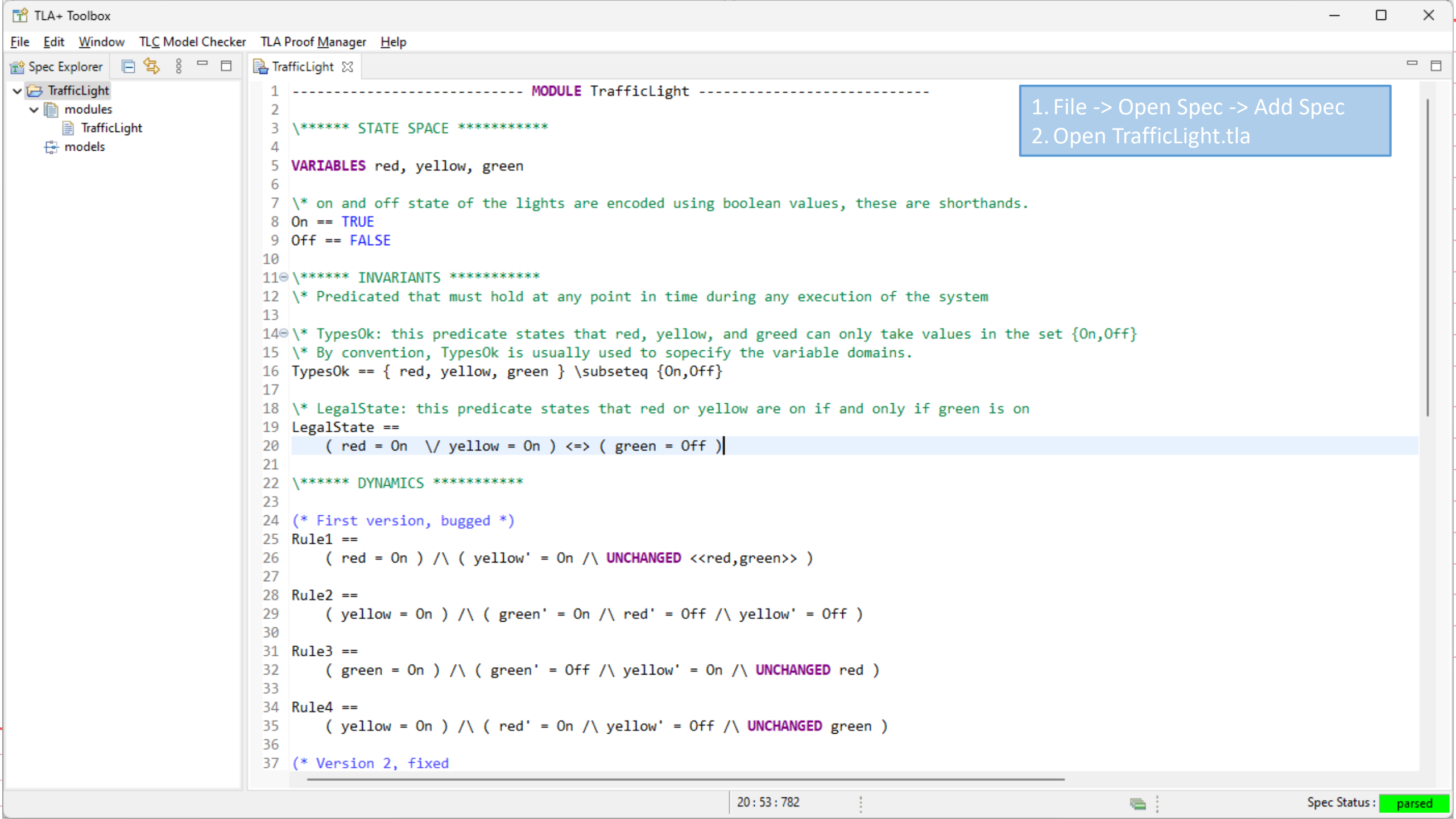
Example: a traffic light controller in TLA+ (4)

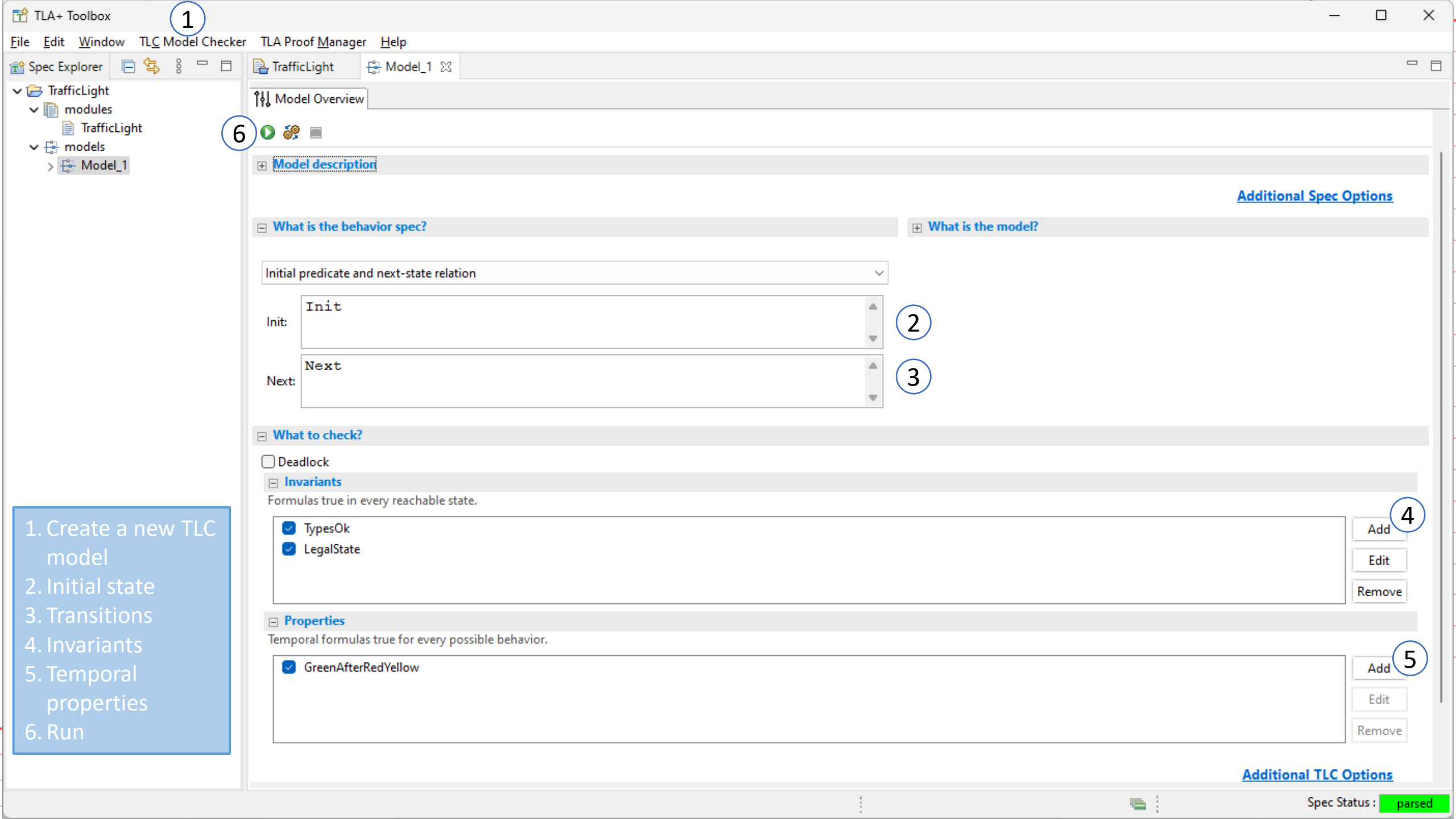
```
\* Temporal Properties

\* GreenAfterRedAndYellow: at any point in time, if red and yellow are on,
then in the next state green will be on

GreenAfterRedAndYellow ==

   [][( red = On  /\ yellow = On ) => (green' = On)]_<<red,yellow,green>>
```

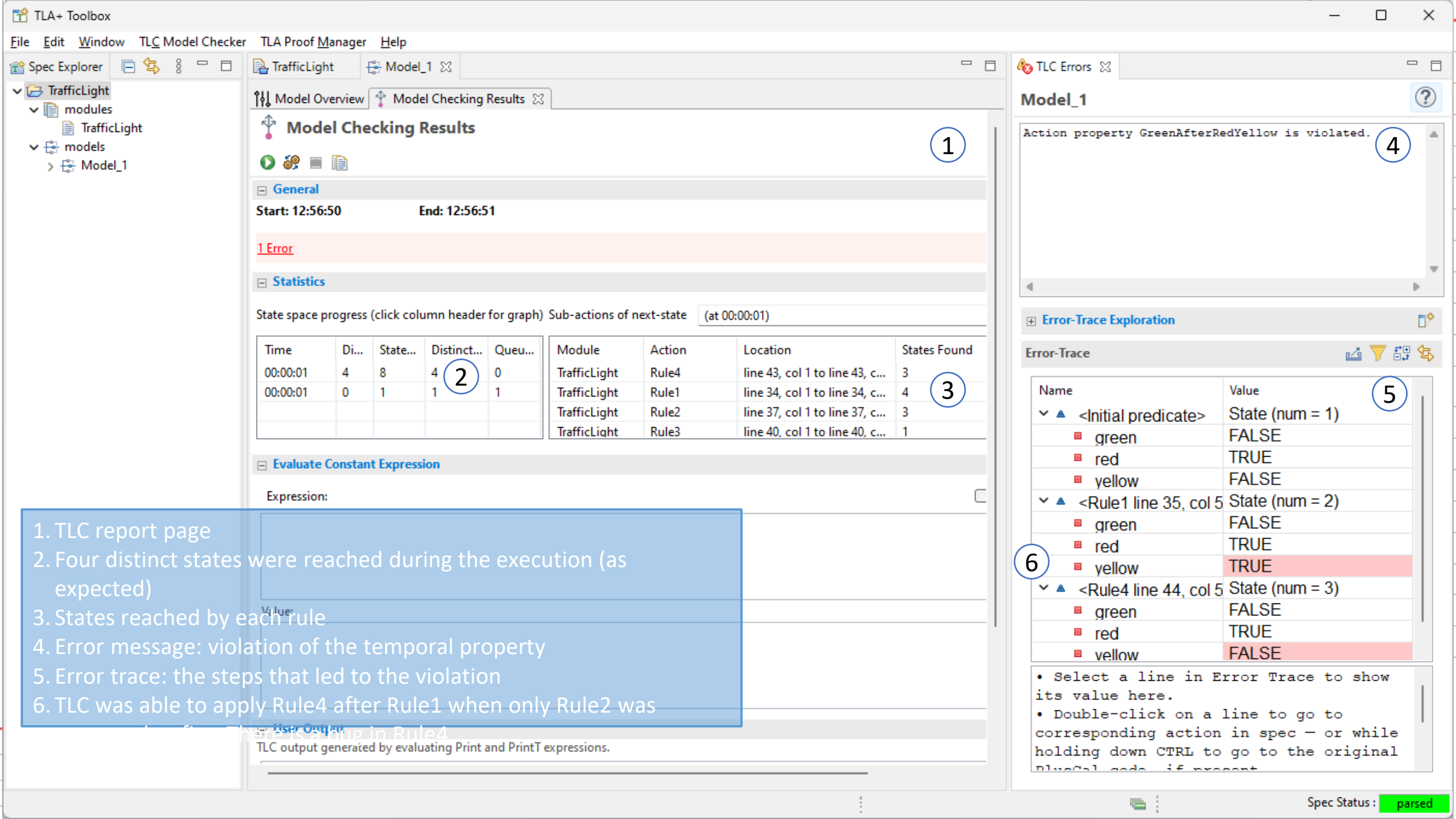
Checking progress requires some care:

- Because of some technical details related to modelling concurrent systems (TLA+ allows stuttering to model certain schedulers), formulas like the one above implicitly include a disjunct “ `\/ UNCHANGED vars`”.
- This means that `[][~UNCHANGED <<red,yellow,green>>]_<<red,yellow,green>>` is a tautology and thus cannot be used to check if our traffic light gets stuck on a state.
- TLC can check for deadlocks (for TLA, states without any transitions). This check is not specified as a temporal property but as an option of the TLC model (more in the next slides).
- To find a loop, we need to selectively remove rules and check that this results in a deadlock (without that rule, the light is stuck).
- If TLC does not detect a deadlock, then some of the other rules are causing an undesired loop.

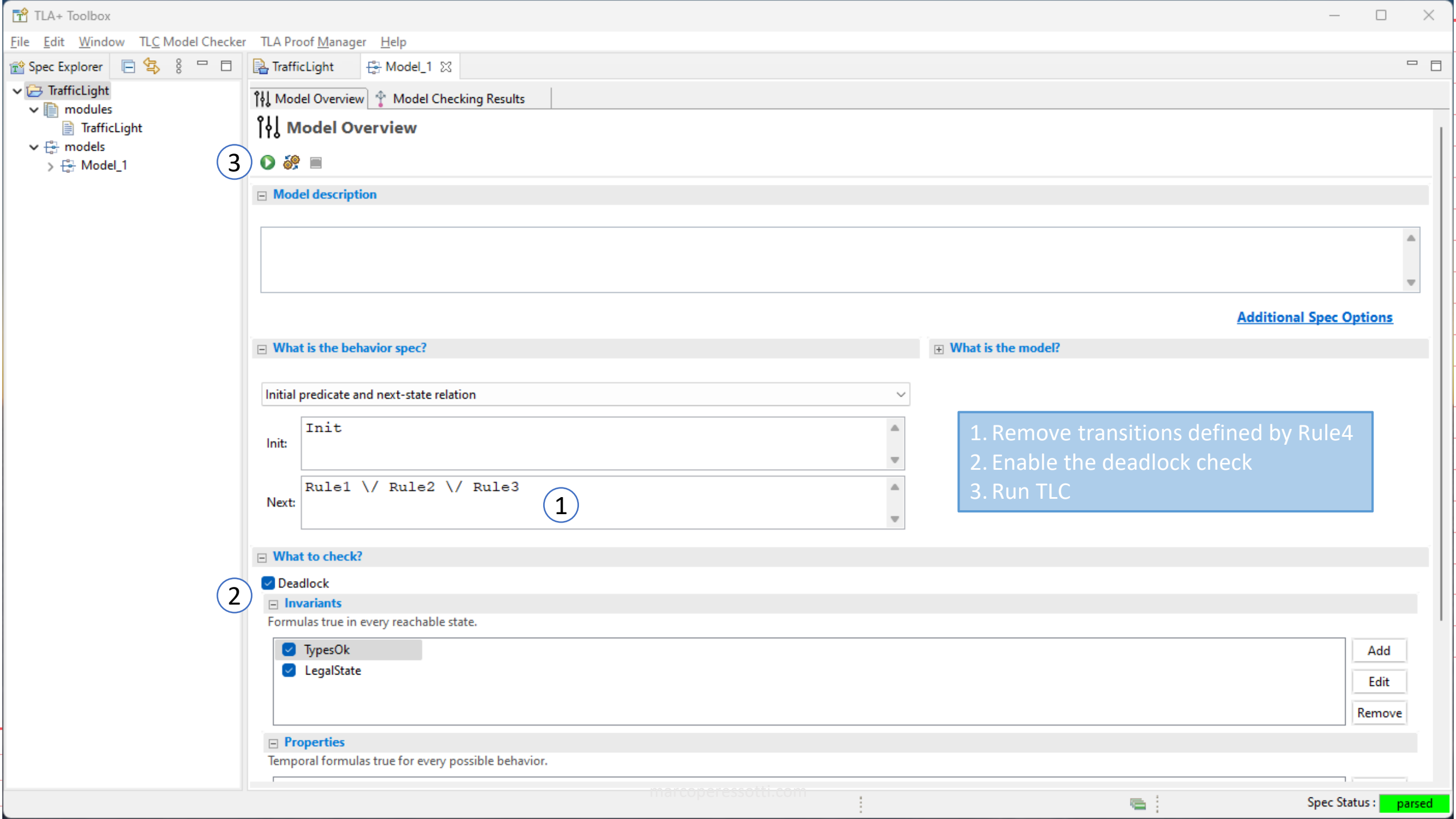
You can find the complete specification of the traffic light at https://bit.ly/gamess-tla


TLA+ Toolbox
File Edit Window TLC Model Checker TLA Proof Manager Help
Spec Explorer
TrafficLight
modules
TrafficLight
models
TrafficLight
1. File -> Open Spec -> Add Spec
2. Open TrafficLight.tla
---------- MODULE TrafficLight ----------
****** STATE SPACE ***********
VARIABLES red, yellow, green
* on and off state of the lights are encoded using boolean values, these are shorthands.
On == TRUE
Off == FALSE
****** INVARIANTS ***********
* Predicated that must hold at any point in time during any execution of the system
* TypesOk: this predicate states that red, yellow, and greed can only take values in the set {On,Off}
* By convention, TypesOk is usually used to sopecify the variable domains.
TypesOk == { red, yellow, green } \subseteq {On,Off}
* LegalState: this predicate states that red or yellow are on if and only if green is on
LegalState ==
(red = On \/ yellow = On) <=> (green = Off)
****** DYNAMICS ***********
(* First version, bugged *)
Rule1 ==
(red = On) /\ (yellow' = On /\ UNCHANGED <<red,green>>)
Rule2 ==
(yellow = On) /\ (green' = On /\ red' = Off /\ yellow' = Off)
Rule3 ==
(green = On) /\ (green' = Off /\ yellow' = On /\ UNCHANGED red)
Rule4 ==
(yellow = On) /\ (red' = On /\ yellow' = Off /\ UNCHANGED green)
(* Version 2, fixed
20 : 53 : 782
Spec Status : parsed


TLA+ Toolbox
1
File Edit Window TLC Model Checker TLA Proof Manager Help
Spec Explorer
TrafficLight
modules
TrafficLight
models
Model_1
TrafficLight
Model_1
Model Overview
6
Model description
Additional Spec Options
What is the behavior spec?
What is the model?
Initial predicate and next-state relation
Init:
Init
2
Next:
Next
3
What to check?
Deadlock
Invariants
Formulas true in every reachable state.
TypesOk
LegalState
Add
Edit
Remove
4
Properties
Temporal formulas true for every possible behavior.
GreenAfterRedYellow
Add
Edit
Remove
5
Additional TLC Options
Spec Status : parsed
1. Create a new TLC model
2. Initial state
3. Transitions
4. Invariants
5. Temporal properties
6. Run



1. TLC report page
2. Four distinct states were reached during the execution (as expected)
3. States reached by each rule
4. Error message: violation of the temporal property
5. Error trace: the steps that led to the violation
6. TLC was able to apply Rule4 after Rule1 when only Rule2 was


TLA+ Toolbox
File Edit Window TLC Model Checker TLA Proof Manager Help
Spec Explorer
TrafficLight
modules
TrafficLight
models
Model_1
TrafficLight
Model_1
Model Overview
Model Checking Results
Model Overview
3
Model description
Additional Spec Options
What is the behavior spec?
What is the model?
Initial predicate and next-state relation
Init:
Init
Next:
Rule1 \/ Rule2 \/ Rule3
1
1. Remove transitions defined by Rule4
2. Enable the deadlock check
3. Run TLC
What to check?
2
Deadlock
Invariants
Formulas true in every reachable state.
TypesOk
LegalState
Add
Edit
Remove
Properties
Temporal formulas true for every possible behavior.
Spec Status : parsed

TLA+ Toolbox

File Edit Window TLC Model Checker TLA Proof Manager Help

Spec Explorer

- TrafficLight
  - modules
    - TrafficLight
  - models
    - Model_1

TrafficLight | Model_1

Model Overview | Model Checking Results

Model Checking Results

General

Start: 13:15:58 **End: 13:15:58** **Not running**

Fingerprint collision probability: calculated: 4.3E-19

Statistics

State space progress (click column header for graph)

Time	Diame...	States Fou...	Distinct States	Queue Size
00:00:00	4	6	4	0
00:00:00	0	1	1	1

Sub-actions of next-state (at 00:00:00)

Module	Action	Location	States Found	Distinct States
TrafficLight	Rule1	line 34, col 1 to line 34, col 5	2	1
TrafficLight	Rule2	line 37, col 1 to line 37, col 5	2	1
TrafficLight	Rule3	line 40, col 1 to line 40, col 5	1	1
TrafficLight	Init	line 70, col 1 to line 70, col 4	1	1

Evaluate Constant Expression

Expression: No Behavior Spec

Value:

User Output

TLC output generated by evaluating Print and PrintT expressions.

```
No output is available
```

TLC does not find deadlocks, there is a loop despite removing Rule4

Spec Status : parsed

TLA+ Toolbox

File Edit Window TLC Model Checker TLA Proof Manager Help

Spec Explorer

- TrafficLight
  - modules
    - TrafficLight
  - models
    - Model_1

TrafficLight | Model_1

Model Overview | Model Checking Results

Model Overview

Model description

Additional Spec Options

What is the behavior spec?

Initial predicate and next-state relation

Init: `Init`

Next: `Rule1`

What is the model?

1. Remove transitions defined by Rule3
2. Enable the deadlock check
3. Run TLC
4. No deadlocks, loop in Rule1 and 2
5. Remove transitions defined by Rule2
6. No deadlocks, loop in Rule1

What to check?

Deadlock

Invariants

Formulas true in every reachable state.

- TypesOk
- LegalState

Add | Edit | Remove

Properties

Temporal formulas true for every possible behavior.

Spec Status : parsed

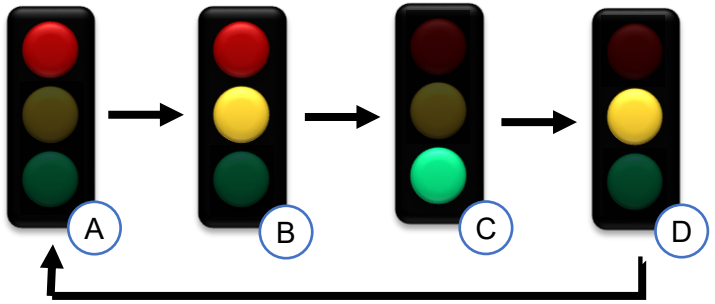
Example: a traffic light controller in TLA+ (fixed)

```
\* Same variables, invariants and temporal properties
\* […]
\* Dynamics (version 2)
Rule1 == ( red = On /\ yellow = Off)
      /\ ( yellow' = On /\ UNCHANGED <<red,green>> )
Rule2 == ( yellow = On )
      /\ ( green' = On /\ red' = Off /\ yellow' = Off )
Rule3 == ( green = On )
      /\ ( green' = Off /\ yellow' = On /\ UNCHANGED red )
Rule4 == ( red = Off /\ yellow = On)
      /\ ( red' = On /\ yellow' = Off /\ UNCHANGED green )
Next == Rule1 \/ Rule2 \/ Rule3 \/ Rule4
Init == red = On /\ yellow = Off /\ green == Off
```

TLA+ spec available at https://bit.ly/gamess-tla

Example: a traffic light controller (fixed)

Desired behaviour



Implementation (v2)

State (variables)

- Lights (on/off)
- Timer

Dynamics (actions)

1. If wait then
2. If wait then
3. If wait then
4. If wait then

Fixes

Model

State (variables)

- Lights (on/off)

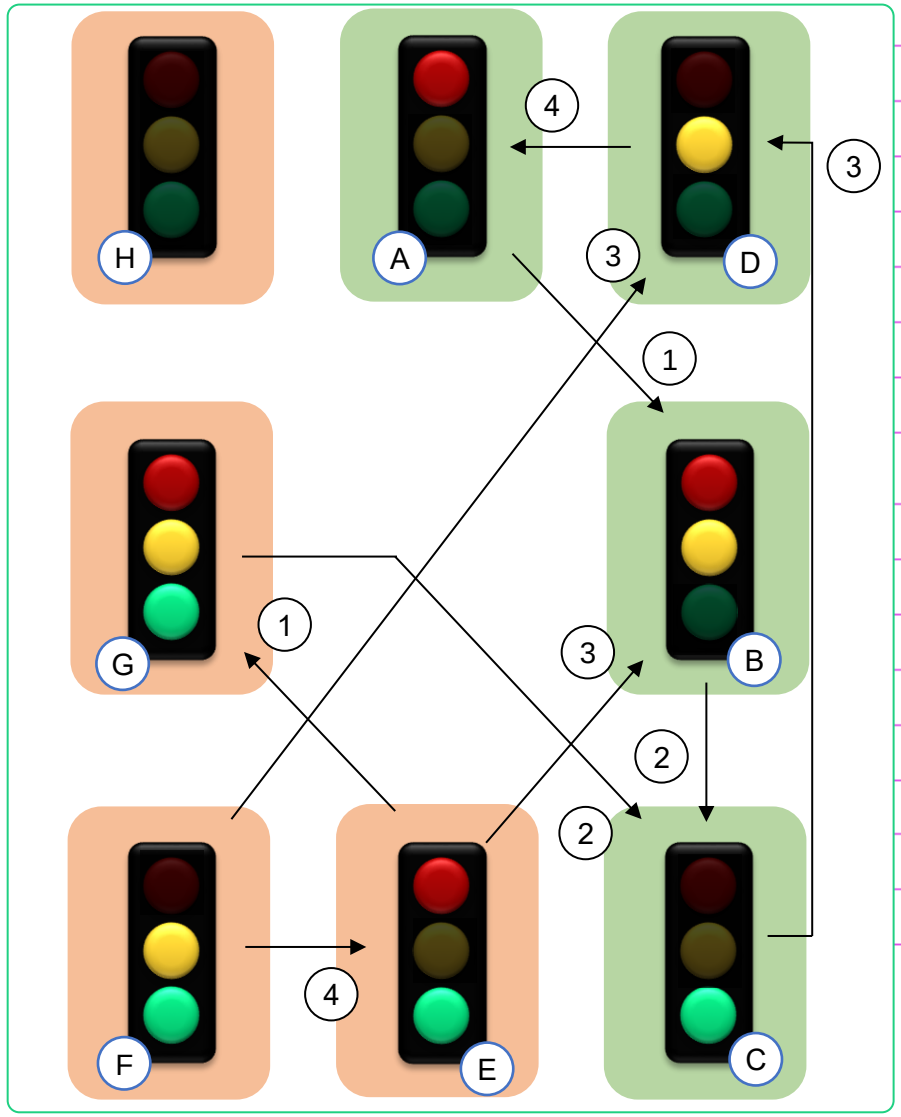
Dynamics (actions)

1. If then
2. If then
3. If then
4. If then

Fixes

Properties

- Never stops
- Cannot reach an illegal state from a legal one
- after
- ….



Learning TLA+

The traffic light example is a first step into using TLA+ and is meant to provide an intuition on the approach of studying systems using formal specifications.

The remainder of this deck contains a few examples about verifying simple sequential and concurrent programs.

You can find the source for these examples at https://bit.ly/gamess-tla together with detailed comments and explanations of the code.

Before approach these, especially those on concurrent programs, it might be beneficial to go through the first 5 chapters of TLA+ video course https://lamport.azurewebsites.net/video/videos.html

Example: array search

This example illustrates a fundamental approach: model the problem before the program.

1. Download the material at https://bit.ly/gamess-tla
2. Model the problem (see ArraySearch.tla)
    1. We start by defining the domain of the problem: we define arrays
    2. We define the problem: we define “correct search result”
    3. We define an “oracle” that leverages TLA to get a correct solution given an input
    4. We verify that 2 and 3 are in agreement to build confidence in the model of the problem before moving to the algorithmic solutions (the programs)
3. Model the algorithmic solution aka the problem (see LinearSearch.tla)
    1. Extend the module that model the problem to use its basic definitions (ArraySearch)
    2. Implement the program (linear search)
    3. Verify that the program terminates returning a correct result (see item 1.4)
4. As an exercise, model Binary Search (for a solution, see BinarySearch.tla)
    1. Extend ArraySearch to use its basic definitions
    2. Refine the model of the problem (binary search assumes that the array to search in is sorted)
    3. Implement binary search
    4. Verify that binary search terminates returning a correct search result

Example: mutual exclusion in PlusCal/TLA+

Guarantee mutually exclusive access to a resource shared by a set of processes

Hyman algorithm (1966)

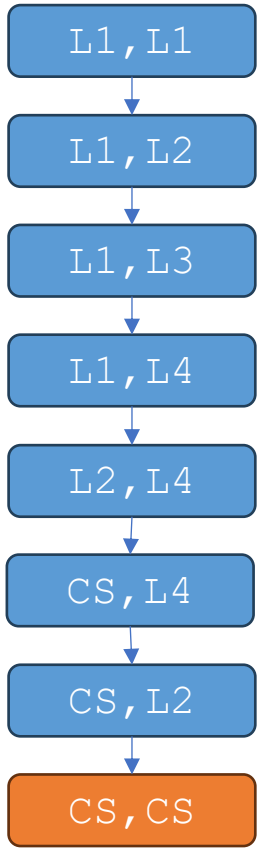
- Shared variables b_1, b_2, k
- Each process runs the code below
- (i denotes its id, j the other proc)

```
while true do
begin
    'noncritical section';
    b_i := true;
    while k ≠ j do begin
                      while b_j do skip;
                      k := i
                  end;
    'critical section';
    b_i := false;
end
```

- There is a bug…

```
\* processes
Procs == {1,2}

(* --algorithm MutEx {
    variables b = [i \in Procs |-> FALSE],
              k \in Procs;
    define { MutualExclusion ==
      []~(\forall i \in Procs : pc[i] = "CS")
    }
    process (p \in Procs) {
        L1: while (TRUE) {
            b[self] := TRUE;
            L2: while (k /= self) {
                L3: while (b[k]) { skip; };
                L4: k := self;
            };
            (* critical section*)
            CS: skip;
            (* end critical section*)
            b[self] := FALSE;
        }};
}
*)
```

⚠ Violation



Example: mutual exclusion in PlusCal/TLA+

Guarantee mutually exclusive access to a resource shared by a set of processes

Peterson algorithm (1985)

- Shared variables b_1, b_2, k
- Each process runs the code below
- (i denotes its id, j the other proc)

```
while true do
begin
    'noncritical section';
    b_i := true;
    k := j;
    while (b_j and k = j) do skip;
    'critical section';
    b_i := false;
end
```

- There is no bug

```
\* processes
Procs == {1,2}

(* --algorithm MutEx {
    variables b = [i \in Procs |-> FALSE],
              k \in Procs;
    define { MutualExclusion ==
      []~(\forall i \in Procs : pc[i] = "CS")
    }
    process (p \in Procs) {
        L1: while (TRUE) {
            b[self] := TRUE;
            k := CHOOSE j \in Procs :
                    j /= self;
            L2: while (k /= self /\ b[k]) {
                    skip; };
            (* critical section*)
            CS: skip;
            (* end critical section*)
            b[self] := FALSE;
        }};
}
*)
```

✅ No violations

Find out more

https://gamess.dk/

GameSS

Who is behind GameSS

Partners behind the project

SDU

DTU

IT UNIVERSITY OF CPH

AALBORG UNIVERSITET

Happy 42

Collaborators

ERHVERVSAKADEMI AARHUS

kea KØBENHAVNS ERHVERVSAKADEMI

Dania ERHVERVSAKADEMI

UCN

UCL Erhvervsakademi og Professionshøjskole

Supported by

Uddannelses- og Forskningsministeriet

Ministry of Higher Education and Science Denmark