Introducing formal methods through gamification

Marco Peressotti

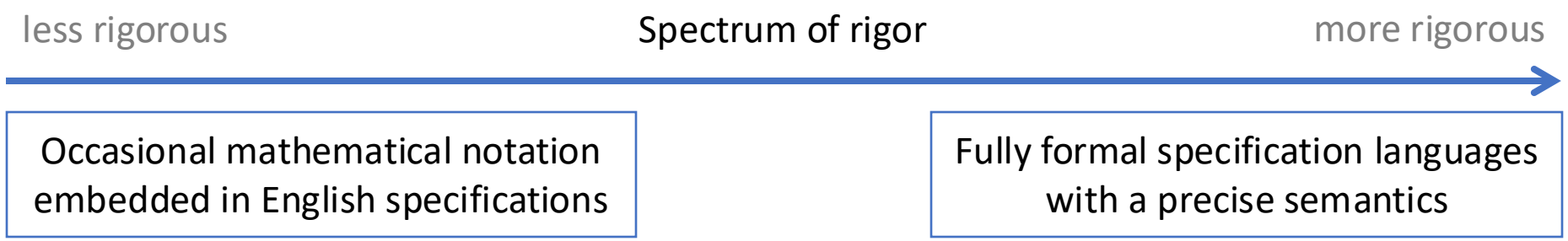
University of Southern Denmark

Workshop on CTF & Offensive Security for Risk Management Awareness

Bologna 2025-12-12

Formal Methods is a HUGE umbrella term

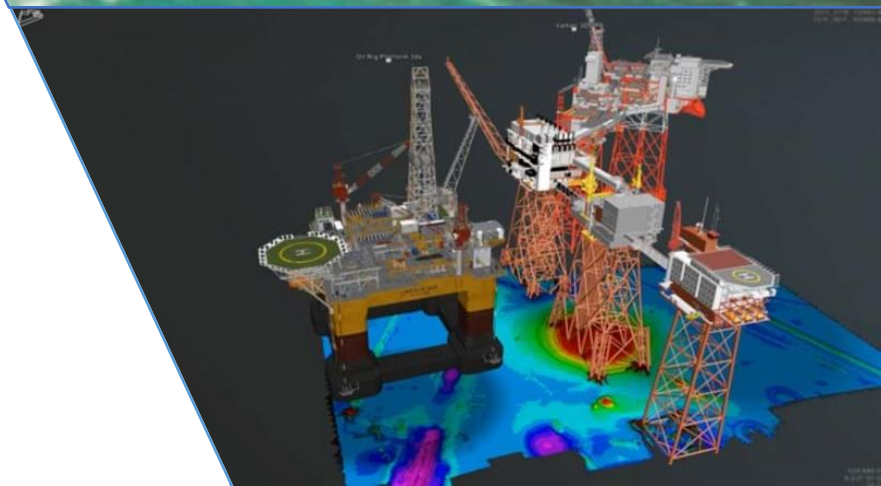
- In general, the use of mathematically rigorous techniques in software and hardware engineering,
- Can assume various forms and levels of rigor



- Broad variety of fundamentals of theoretical computer science:
  - Logics, Formal languages/automata theory, Program semantics, Type theory, ...

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,…)
- Part of certifications



Software is infrastructure: failures, successes, costs, and the case for formal verification.

Case	Sector	Human Cost	Economic Cost	Summary
Therac-25	Healthcare	Yes (6 deaths)	Unquantified	Radiation therapy software delivered hazardous doses due to race conditions.
London Ambulance System	Healthcare	Yes (20-30 deaths)	1.5 million GBP	Ambulance dispatch system collapsed due to memory leak.
Sleipner A Platform	Engineering	No	700 million USD	Modelling error in finite element analysis underestimated structural stresses resulting in the loss of the platform.
Patriot Failure	Aerospace	Yes (28 deaths)	Unquantified	Clock drift due to truncation error prevented interception.
Boeing Starliner OFT-1	Aerospace	No	410 million USD	Synchronisation of spacecraft and launch vehicle clocks failed due to logical bug prevented planned orbital insertion.
Boeing 737 MAX	Aerospace	Yes (346 deaths)	20 billion USD	Crashes due to single point of failure in fly-by-wire system.
Mars Climate Orbiter	Aerospace	No	327 million USD	Loss of mission due to unit mismatch (imperial vs. metric) in flight control software.
Ariane 5 Flight 501	Aerospace	No	370 million USD	Loss of mission due to float to integer conversion error in flight control software.
Toyota SUA	Automotive	Yes (89 deaths)	1.2 billion USD	Suspected software contribution to unintended acceleration.
4LM Project	Railways	No	1 billion GBP	Complexity of real-time software hindered development.
Horizon IT Scandal	Accounting	Indirect (13 suicides)	1 billion GBP (est.)	Software errors led to 900+ false fraud accusations.
CrowdStrike	IT Operations	No	8 billion USD	Update crash led to global business disruptions.
Facebook DNS	IT Operations	No	150 million USD (est.)	Misconfiguration caused hours-long global outage.
EternalBlue (WannaCry, NotPetya)	IT Operations	Indirect	14 billion USD (est.)	Windows use-after-free vulnerability enabled remote code execution. NSA-developed exploit was leaked and used by ransomware and malware campaigns.
Knight Capital	Finance	No	460 million USD	Error in software operations. One server was not updated.
IRON	Finance	No	2 billion USD (est.)	Infrastructure software contributed to market outage.

When is using FM worth the cost?

The traditional answer

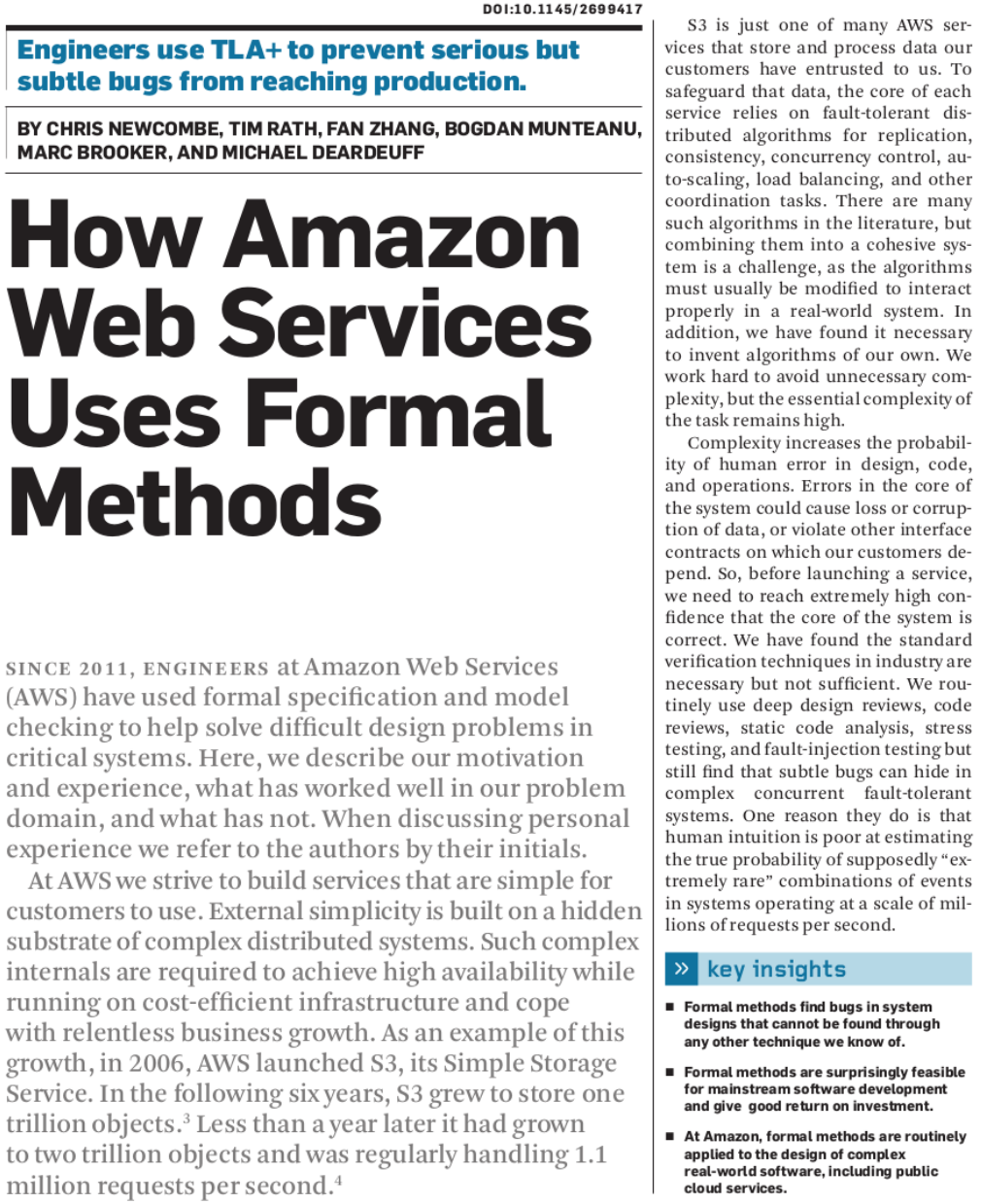
- High-integrity systems
  - Safety critical systems (aerospace, automotive, energy, etc)
  - Security critical systems (banking, voting, etc)
  - Mandated by industry standards (IEC61508, DO-178B)
- Systems where early error detection saves lots of money (e.g. hardware)
- Systems with unpredictable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

Adoption is on the rise thanks to

- 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

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

- Ama
  syste
- In ea
  - fi
    b
  - gi
    a
    c
- Ama
- Engi
  scrat

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

DOI:10.1145/2699417

Engineers use TLA+ to prevent serious but

UNTEANU,

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.

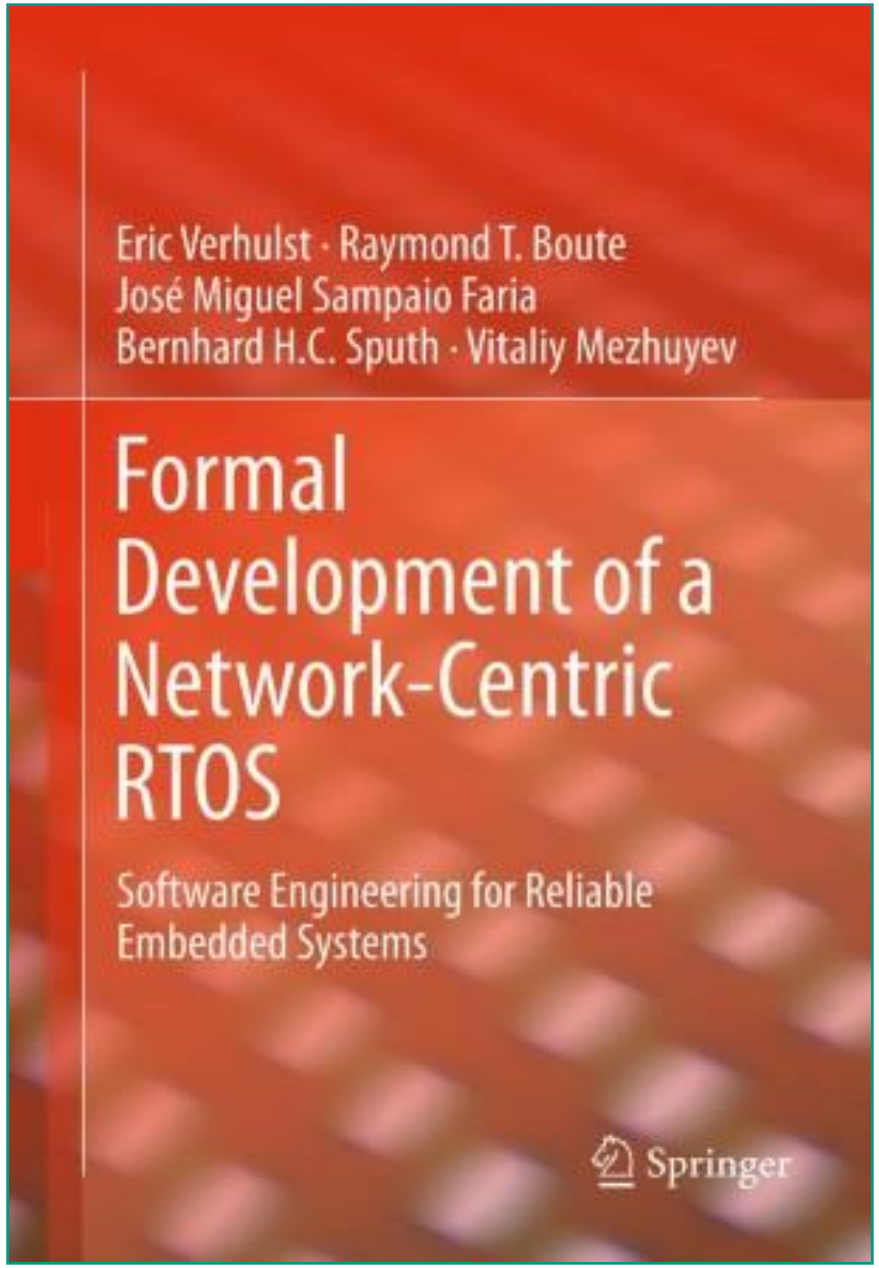
Complexity increases the probability of human error in design, code, and operations. Errors in the core of the system could cause loss or corruption of data, or violate other interface contracts on which our customers depend. So, before launching a service, we need to reach extremely high confidence 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

- 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+ and Model Checking

System

Model
(State Machine)

enjoys

$\models$

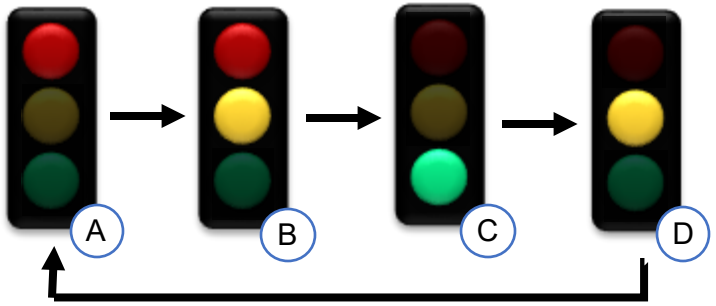
Properties

Formulas
(Temporal Logic)

$$\boldsymbol{G}(green \rightarrow \boldsymbol{X}(yellow\ \boldsymbol{U}\ red))$$

Example: a traffic light controller

Desired behaviour



Implementation (v1)

State (variables)

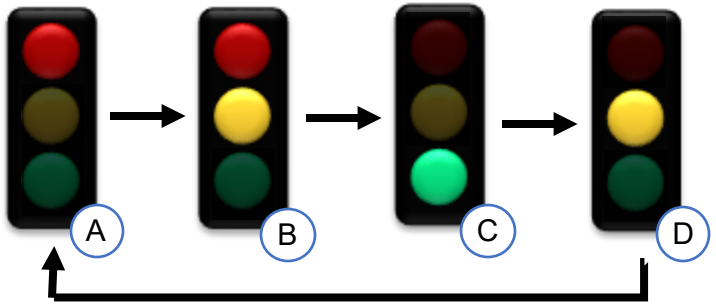
- Lights (on/off)
- Timer

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

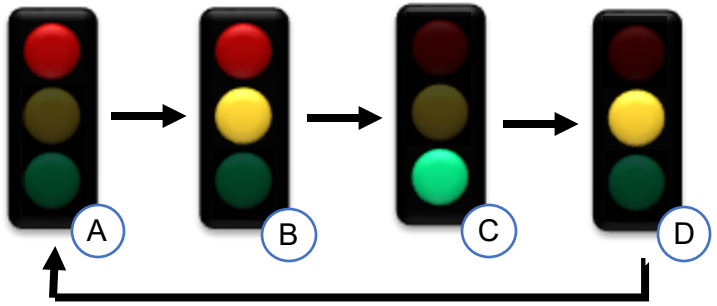
Model

State (variables)

- Lights (on/off)

Example: a traffic light controller

Desired behaviour



Implementation (v1)

State (variables)

- Lights (on/off)
- Timer

Dynamics (actions)

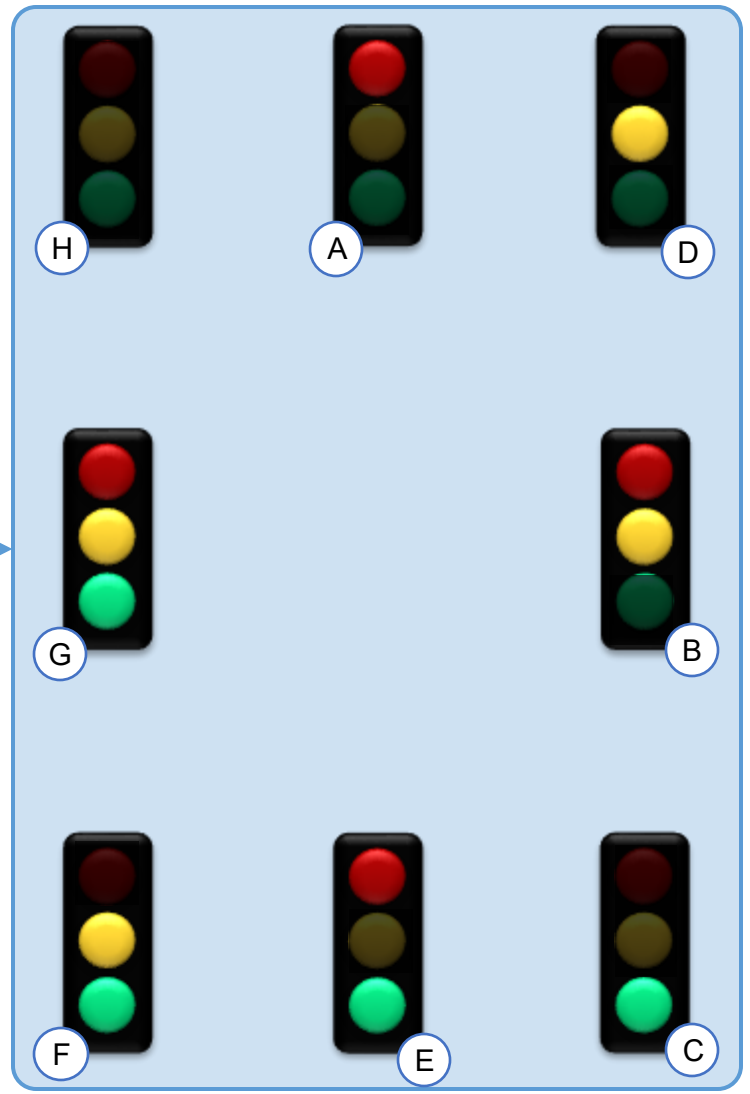
1. If wait then
2. If wait then
3. If wait then
4. If wait then

Model

State (variables)

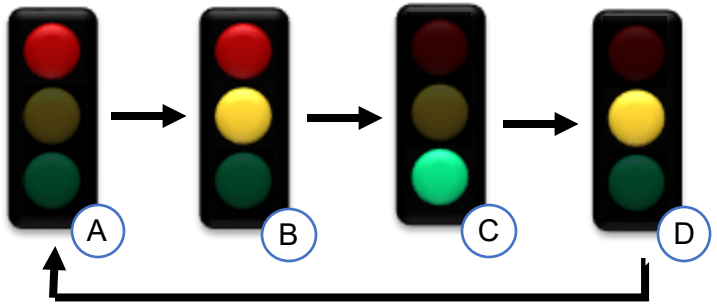
- Lights (on/off)

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)

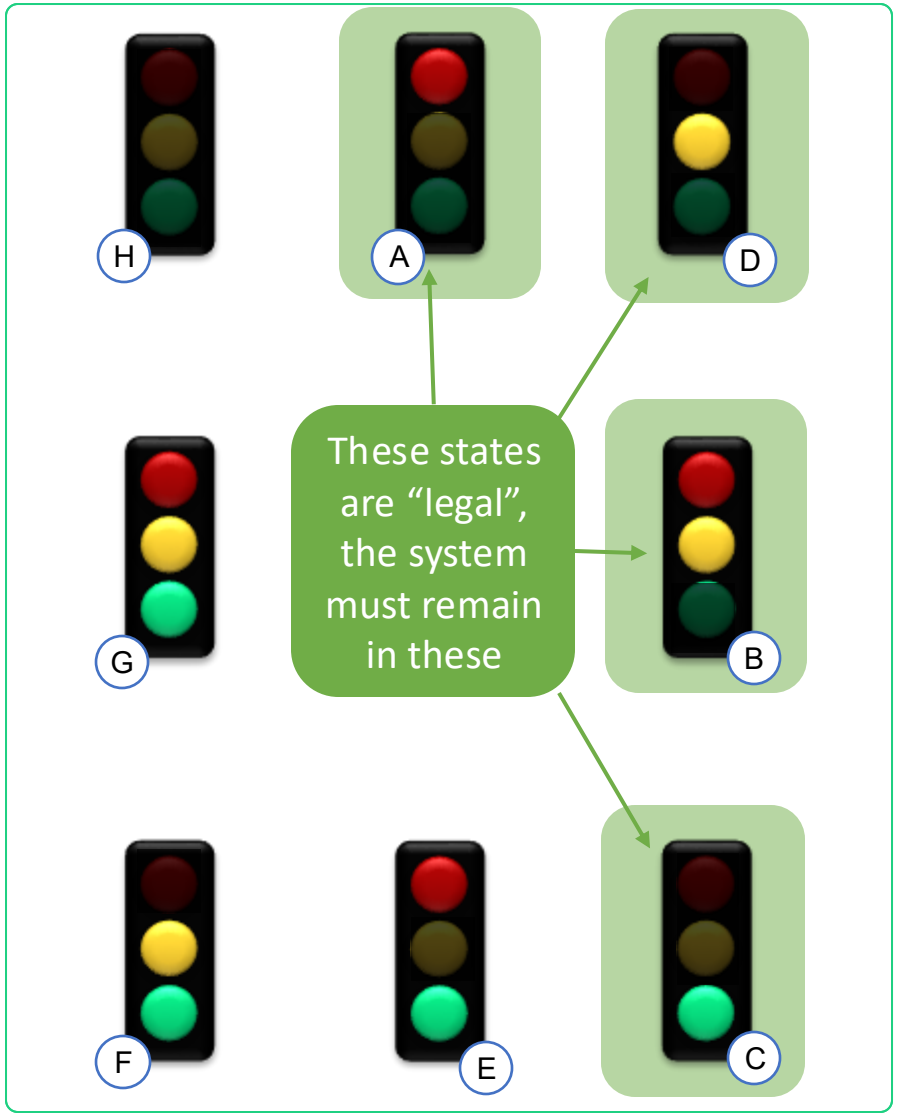
1. If wait then
2. If wait then
3. If wait then
4. If wait then

Model

State (variables)

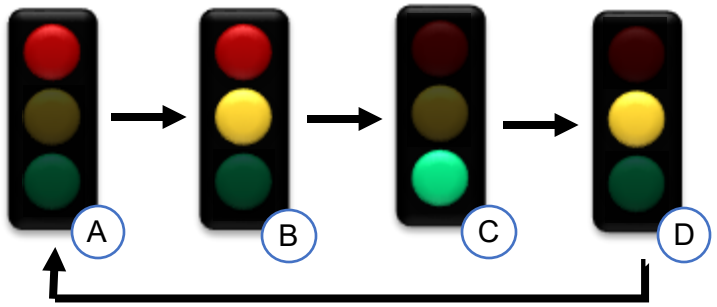
- Lights (on/off)

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

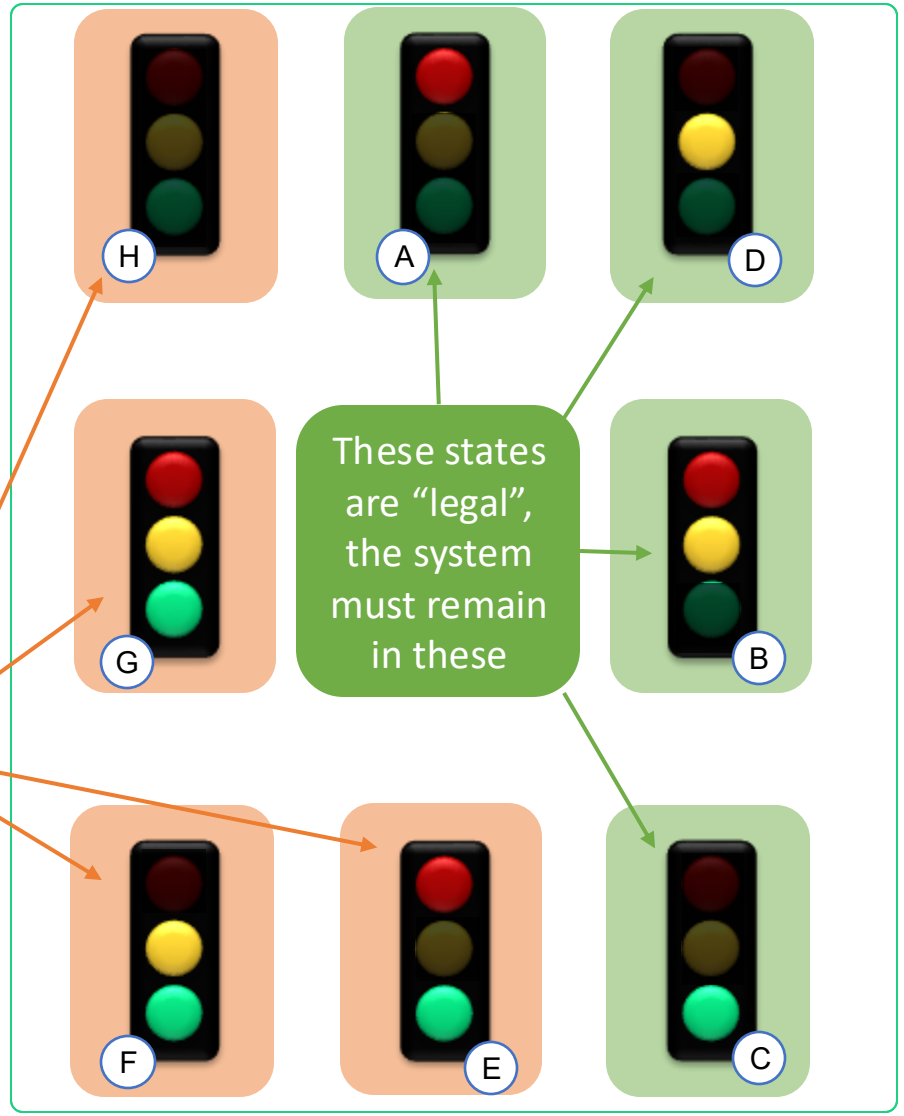
Model

State (variables)

- Lights (on/off)

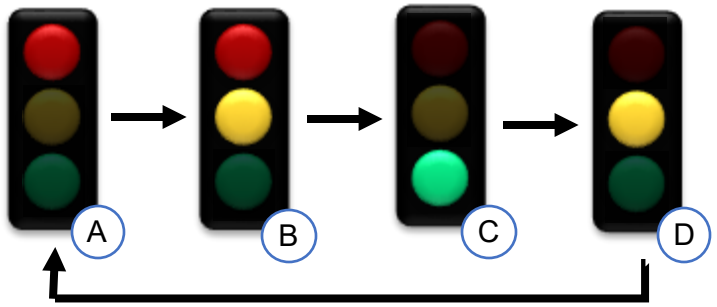
The values that these variables can take, define 8 states (the model does not include time)

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

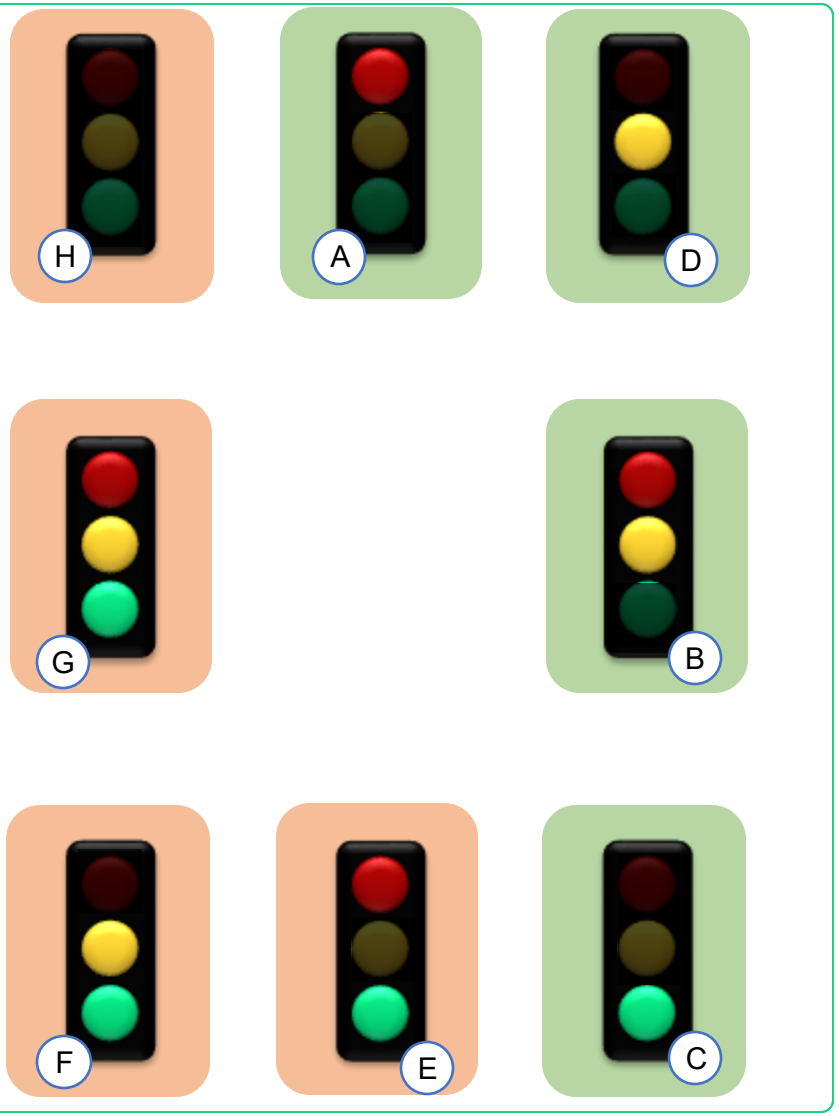
Model

State (variables)

- Lights (on/off)

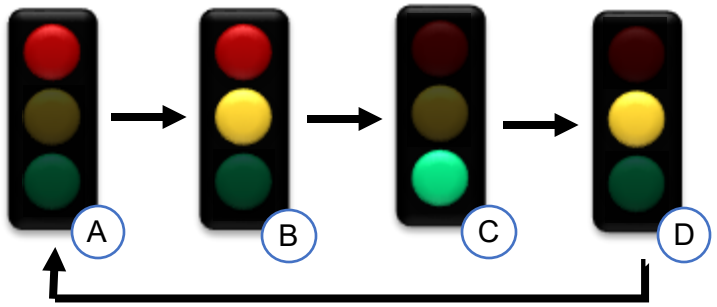
Dynamics (actions)

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)

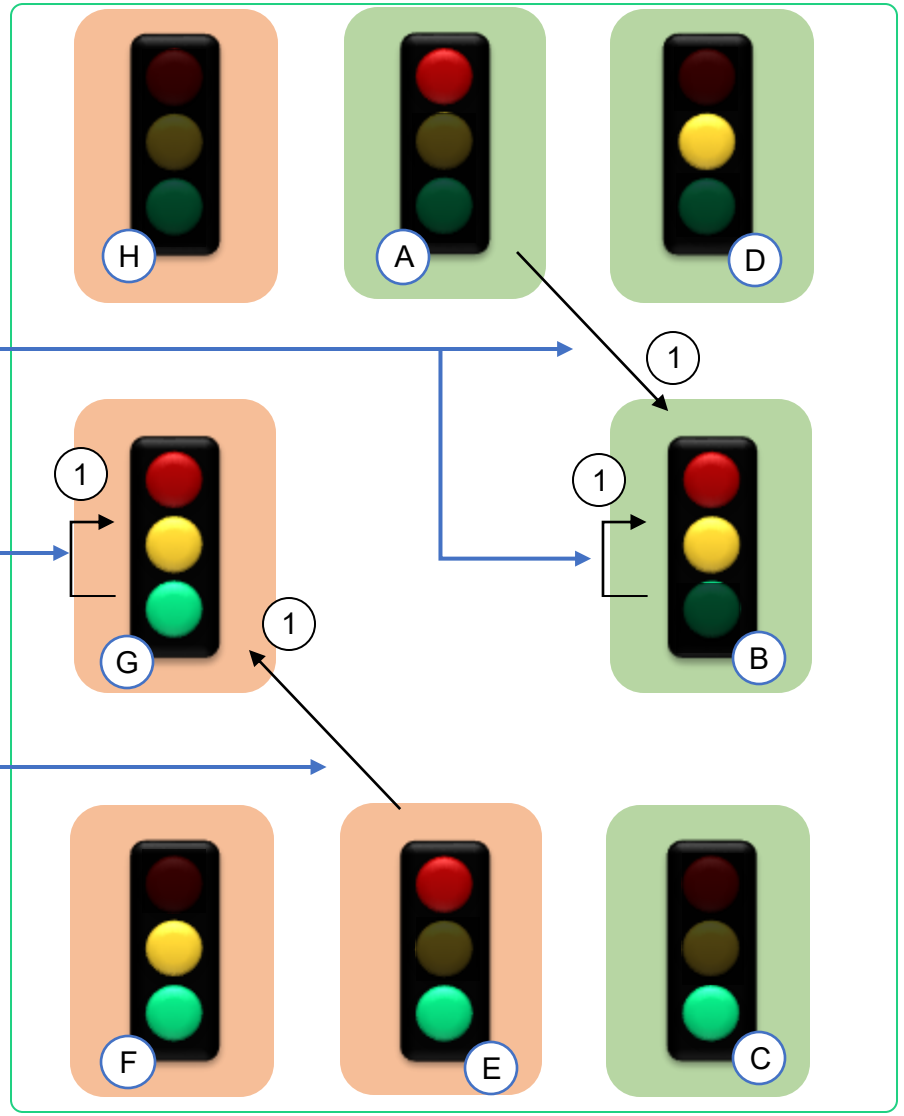
- Lights (on/off)

Dynamics (actions)

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

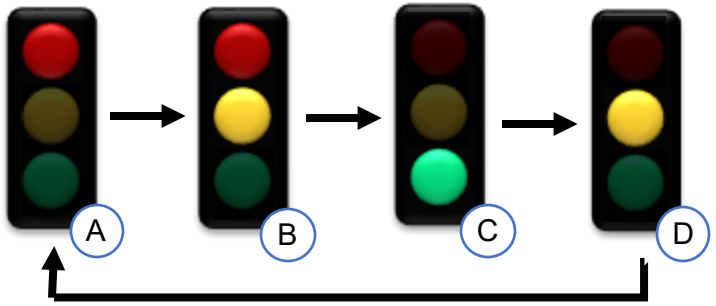
Rule 1: if the red light is on then, turn the yellow light on.

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

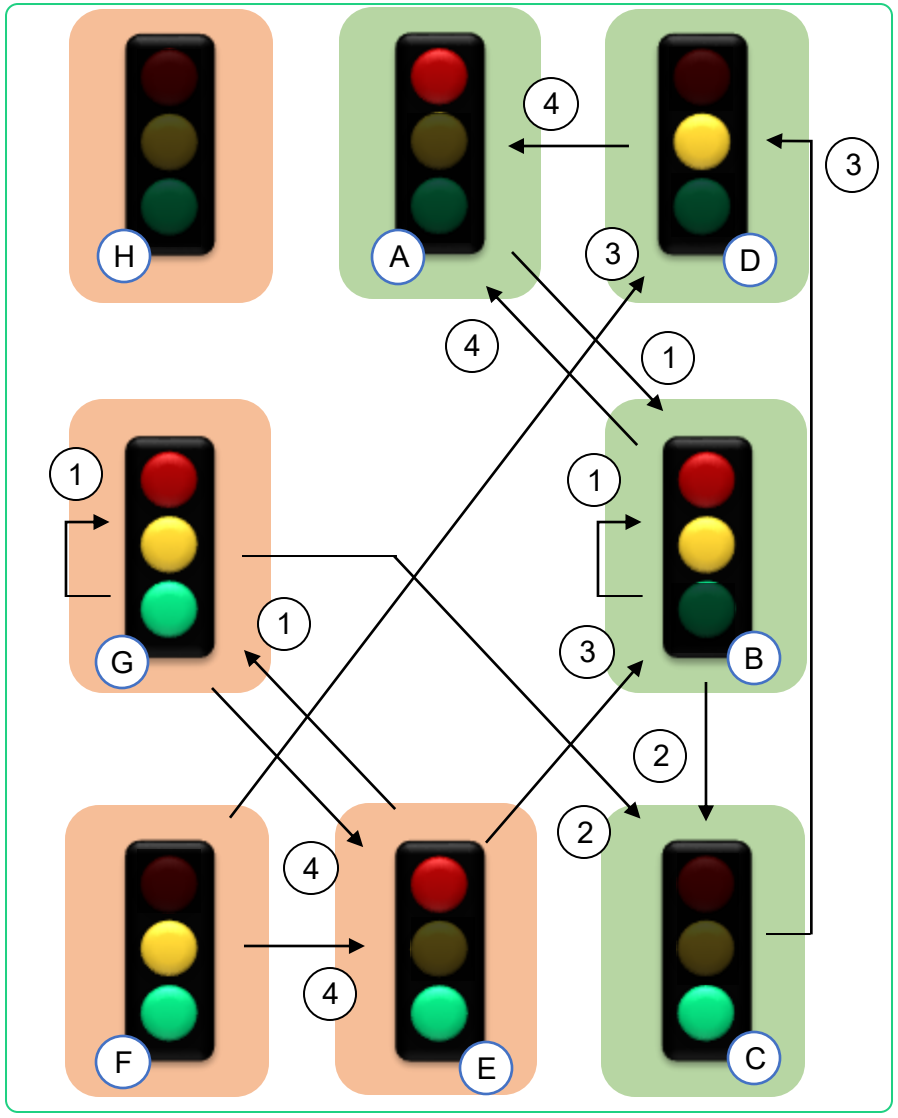
Model

State (variables)

- Lights (on/off)

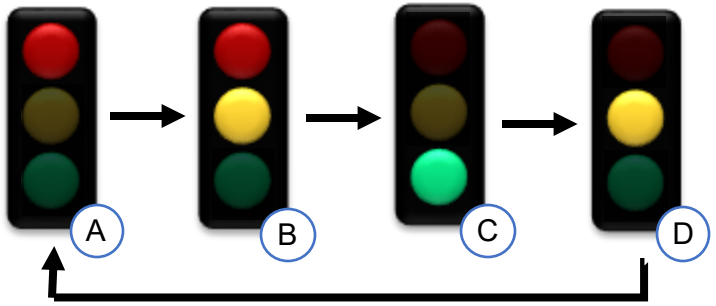
Dynamics (actions)

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)

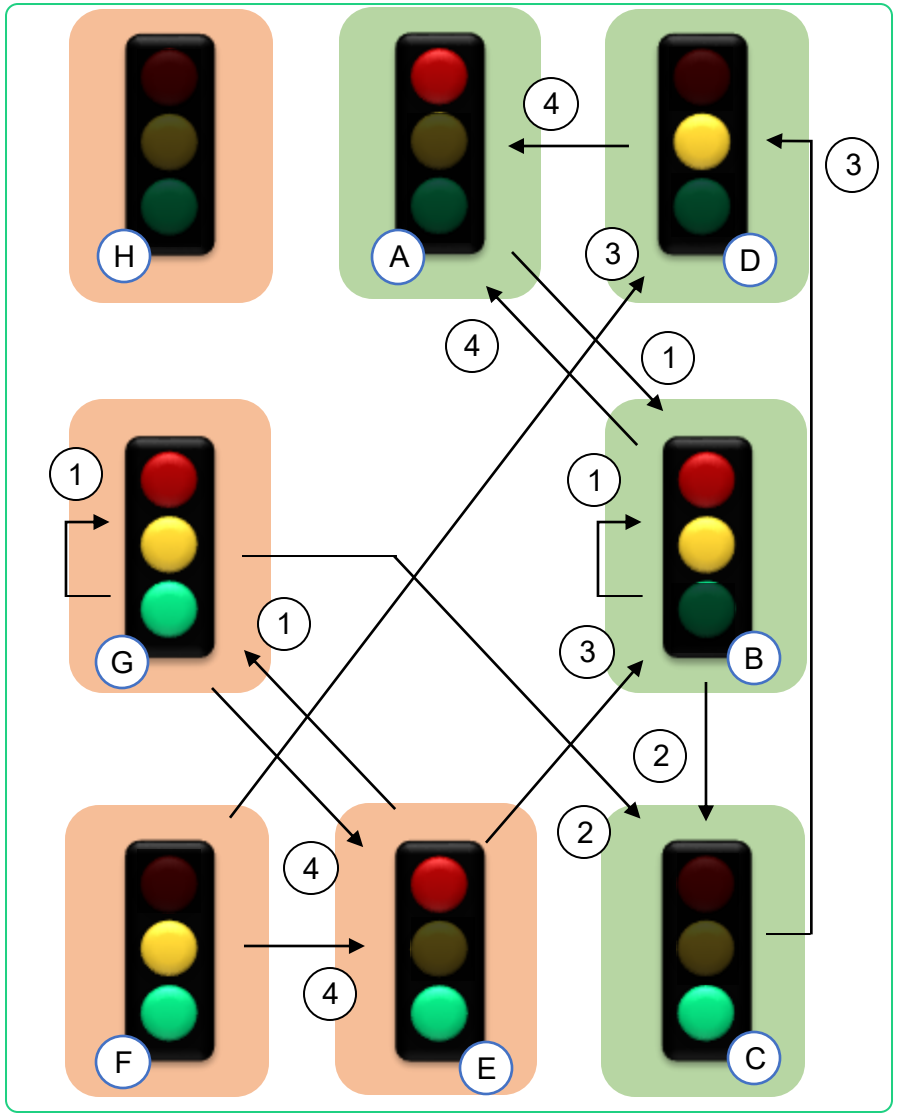
- Lights (on/off)

Dynamics (actions)

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

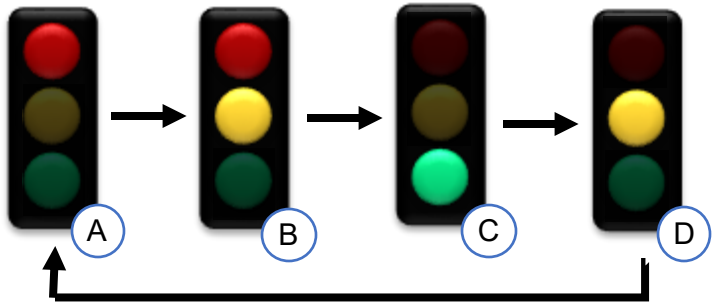
Properties

- Never stay in the same state
- 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)

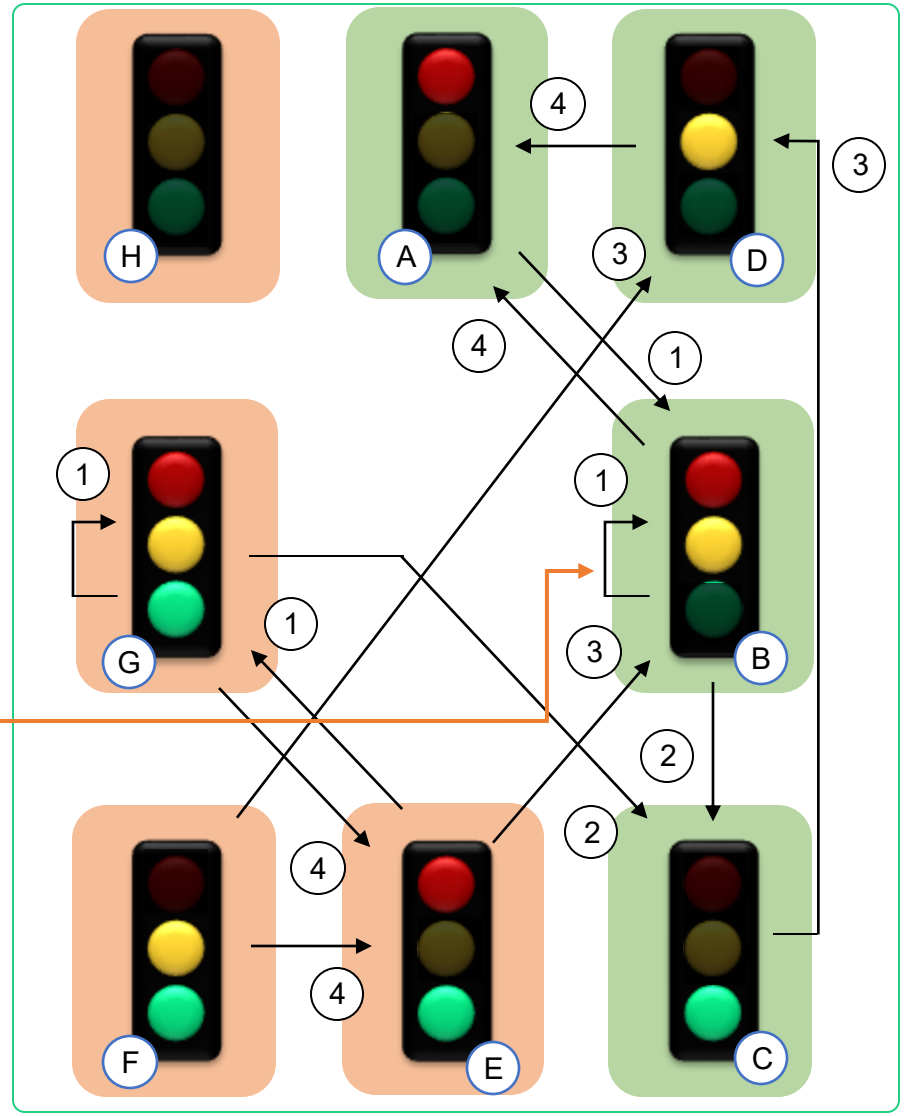
Dynamics (actions)

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

Properties

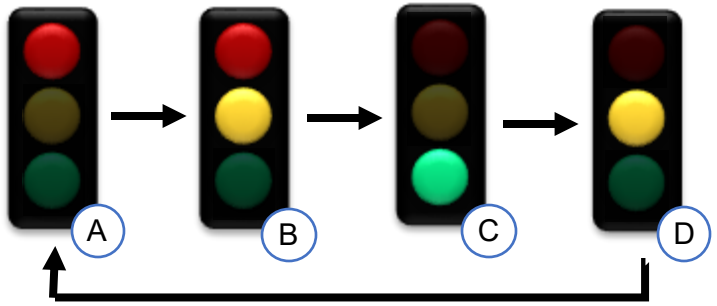
- Never stay in the same state
- Cannot reach an illegal state from a legal one
- after
- ….

This property does not hold



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

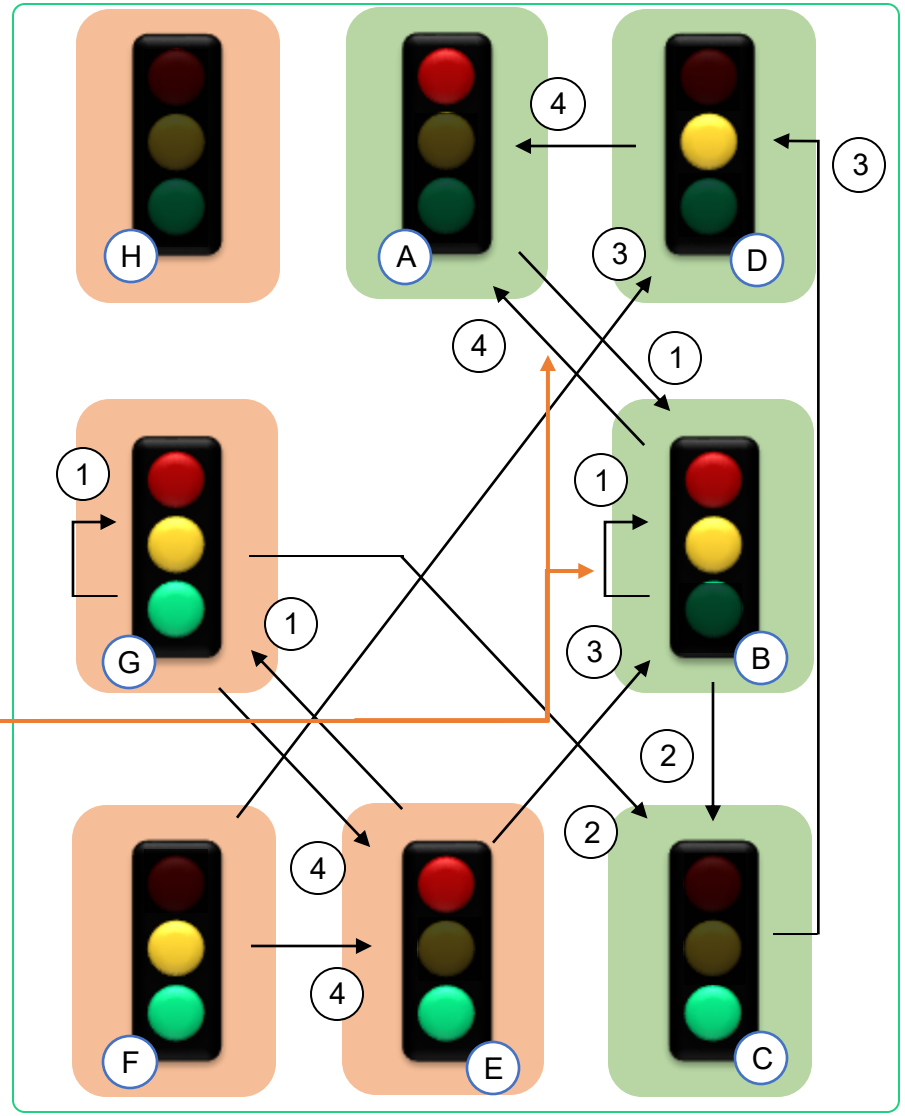
Never stay in the same state

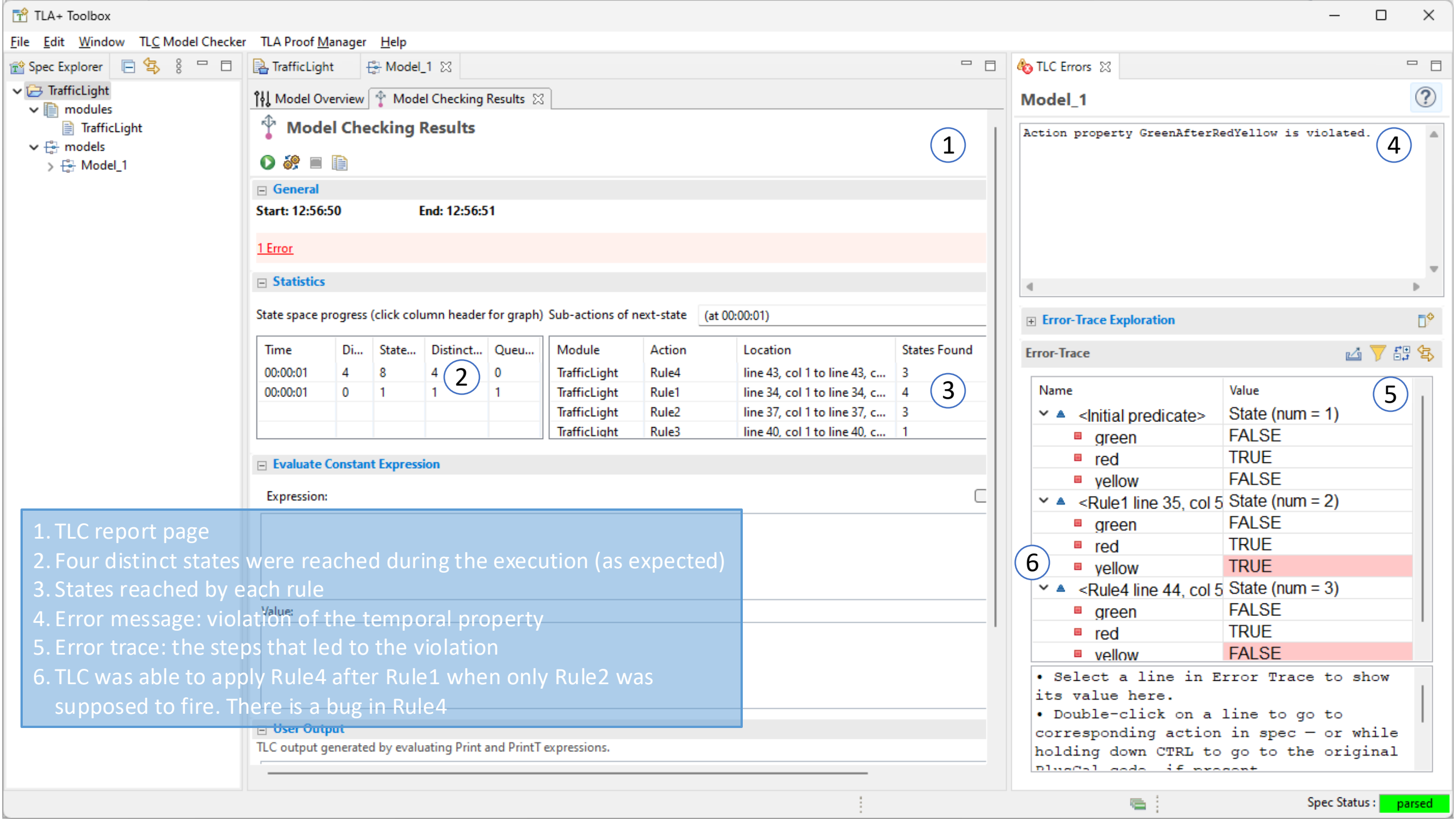
Cannot reach an illegal state from a legal one

after

- ….

This property does not hold

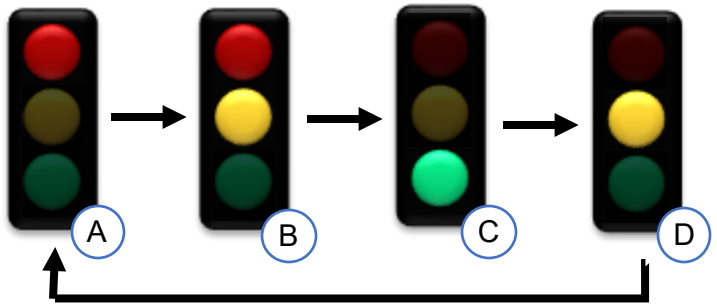




1. TLC report page
2. Four distinct states were reached during the execution (as expected)
3. States reached by each rule
4. Error message: violation of the temporal property
5. Error trace: the steps that led to the violation
6. TLC was able to apply Rule4 after Rule1 when only Rule2 was supposed to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

Example: a traffic light controller (fixed)

Desired behaviour



Implementation (v2)

State (variables)

- Lights (on/off)
- Timer

Fixes

Dynamics (actions)

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

Model

State (variables)

- Lights (on/off)

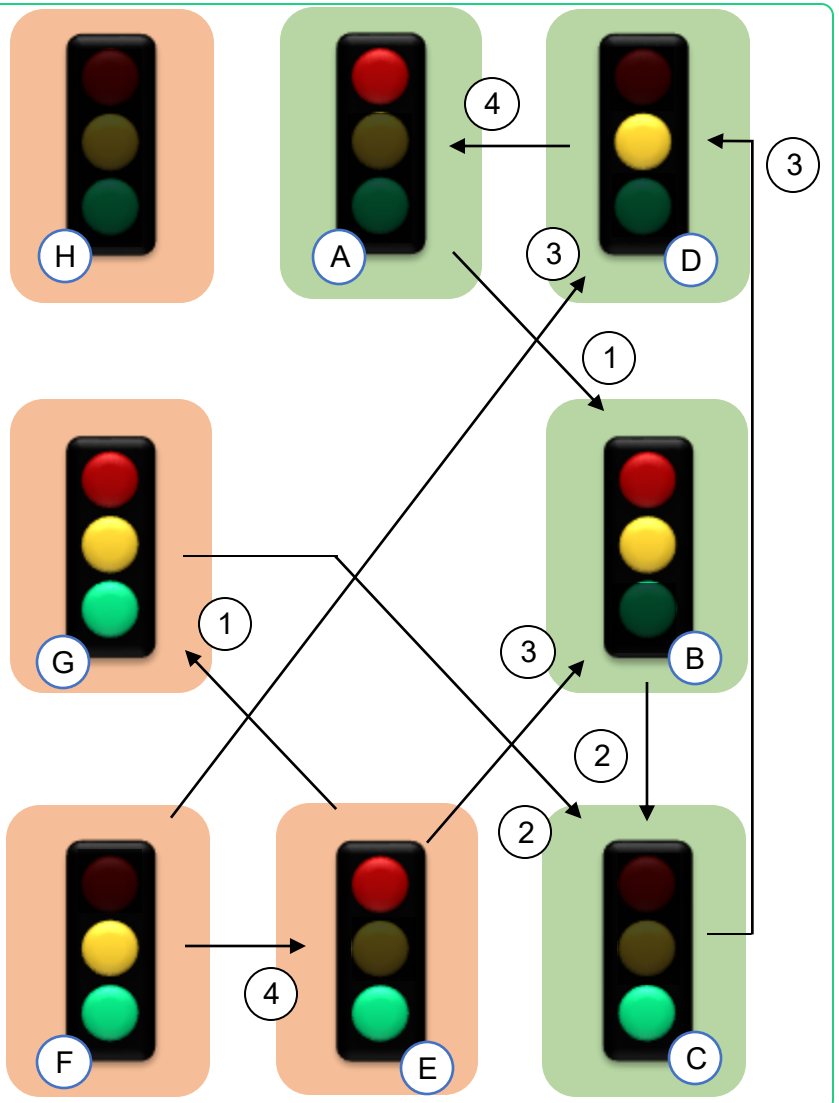
Fixes

Dynamics (actions)

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

Properties

- Never stay in the same state
- Cannot reach an illegal state from a legal one
- after
- ….



Formal Methods belong to CTFs

- Formal methods are not just additional checks or tests, they produce witnesses, counterexamples, violations, invariants, proof certificates.
- These artefacts can map directly to CTF scoring:
  - Find a trace → get a flag.
  - Find an invariant → get a flag.
  - Show a property holds → get a flag.
- They can be part of offensive and defensive scenarios:
  - break the property by producing a violating execution.
  - demonstrate that no unsafe state is reachable.
  - get help in writing or checking patches
- Course exercises can become CTFs (GameSS Project https://gamess.dk/)

Example

[CTF from GameSS Project]

Find a trace where both processes are in the `critical section` at the same time

The algorithm:

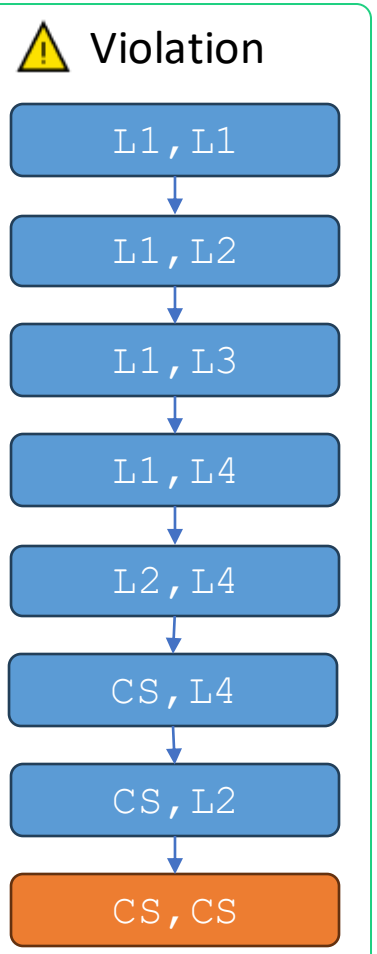
- Shared variables b_1, b_2, k
- Each process runs the code below
- (i denotes its id, j the other proc)

```
L1  while true do
    begin
        'noncritical section';
        b_i := true;
L2      while k ≠ j do begin
L3                      while b_j do skip;
L4                      k := i
                    end;
CS      'critical section';
        b_i := false;
    end
```

(Hyman algorithm, 1966)

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

(* --algorithm MutEx {
    variables b = [i \in Procs |-> FALSE], k \in Procs;
    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;
        }};
    define {
        MutualExclusion ==
           ~(pc[1] = "CS" /\ pc[2] = "CS")
    }
}
*)
```



Example 2

[CTF from GameSS Project]

Show whether both processes can access the `critical section` at the same time

The algorithm:

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

```
L1  while true do
    begin
        'noncritical section';
        b_i := true;
        k := j;
L2      while (b_j and k = j) do skip;
CS      'critical section';
        b_i := false;
    end
```

(Peterson algorithm 1985)

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

(* --algorithm MutEx {
    variables b = [i \in Procs |-> FALSE], k \in Procs;
    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;
        }};
    define {
        MutualExclusion ==
          ~(\forall p \in Procs : pc[p] = "CS")
    }
}
*)
```

✅ No violation

Takeaways from (mostly offensive) CTFs

1. Tie success to producing a witness
   - Flags should only come from a counterexample or proof witness.
2. Witness verification
   - Witnesses must be verified
   - Keep verification costs under control
3. Keep tasks small
   - Assume limited resources (e.g., a laptop)
4. Provide multiple tool paths (in competitions)
   - multiple verification tools (model checkers, constraint solvers etc.),
   - brute force,
   - custom scripts,
   - or even manual reasoning.

What about defensive challenges?

- Formal methods can be used to show the absence of bugs
- A limiting factor is that most formal-methods tools produce proofs that are
  - too large or too opaque,
  - tool-specific and not easily independently validated,
  - difficult to re-check on a CTF platform (but can a CTF trust a solver output?).
- Examples in education
  - Participants submit formal proofs, temporal properties, invariants etc. and the platform checks them
  - Participants submit the proof certificate from the given tool.
- Example in competitions
  - The familiar “patch challenge” designed to make the use of FM advantageous e.g., with large parameter or state spaces, long chains of activation.

Some sources of inspiration

- RERS Challenge
  - Large reactive systems, automatically generated.
  - Challenge: proove reachability of an unsafe state (or more complex LTL properties).
  - Automatic validation of witnesses.
- SV-COMP Verification Tasks
  - Several small programs (usually C or Java).
  - Challange: identify issues related to memory safety, termination, concurrency safety.
  - Automatic scoring of witnesses.
- VerifyThis
  - Very interesting problems and range of tools (often tool independent).
  - However, verification is not fully automated

Introducing formal methods through gamification

Thank you for your attention!

Software is infrastructure: failures, successes, costs and the case for formal verification

The Gamess Project Material on Gamification of Cybersecurity Education