Programm- & Systemverifikation

Temporal Logic and Model Checking

Georg Weissenbacher

184.741

(thanks to Igor Konnov for many slides)

What happened so far

We learned:

- about bugs and assertions
- how to test programs
- how to prove programs correct
- about Bounded Model Checking (BMC)
- how to perform automated reasoning

Today we will learn about (unbounded) Model Checking

Model Checking

Edmund Clarke
Allen Emerson
Joseph Sifakis

Basic idea:

- Assertions in temporal logic
- Programs with finite state space
- *models* instead of programs
- all reachable states are inspected!
- also works for concurrent models

T

$$s \xrightarrow{\mathbf{T}} s'$$

T

$$s \longrightarrow s'$$

$$\langle pc \mapsto 2, x \mapsto 1\rangle \xrightarrow{T} \langle pc \mapsto 3, x \mapsto 2\rangle$$

T

$$s \longrightarrow s'$$

$$\langle pc \mapsto 2, x \mapsto 1 \rangle \xrightarrow{T} \langle pc \mapsto 3, x \mapsto 2 \rangle$$

(T: operational semantics of program or circuit)

T

$$s \longrightarrow s'$$

$$\langle pc \mapsto 2, x \mapsto 1 \rangle \xrightarrow{T} \langle pc \mapsto 3, x \mapsto 2 \rangle$$

(T: operational semantics of program or circuit)

The **Model Checking** problem:

I
"starting states"

$\neg P$
"bad states"

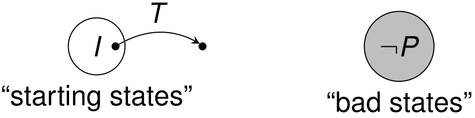
T

$$s \longrightarrow s'$$

$$\langle pc \mapsto 2, x \mapsto 1\rangle \xrightarrow{\;T\;} \langle pc \mapsto 3, x \mapsto 2\rangle$$

(T: operational semantics of program or circuit)

The **Model Checking** problem:



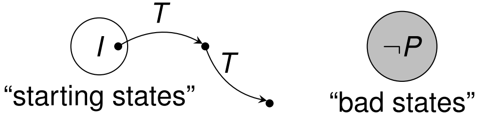
T

$$s \longrightarrow s'$$

$$\langle pc \mapsto 2, x \mapsto 1 \rangle \xrightarrow{T} \langle pc \mapsto 3, x \mapsto 2 \rangle$$

(T: operational semantics of program or circuit)

The **Model Checking** problem:



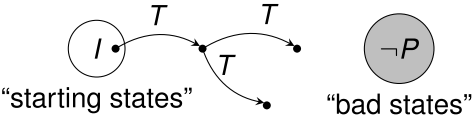
T

$$s \longrightarrow s'$$

$$\langle pc \mapsto 2, x \mapsto 1 \rangle \xrightarrow{T} \langle pc \mapsto 3, x \mapsto 2 \rangle$$

(T: operational semantics of program or circuit)

The **Model Checking** problem:



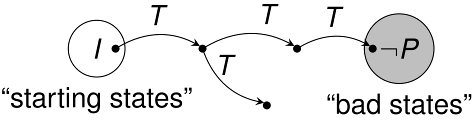
T

$$s \longrightarrow s'$$

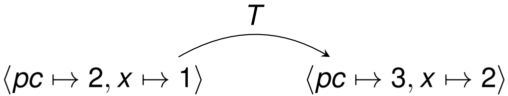
$$\langle pc \mapsto 2, x \mapsto 1 \rangle \xrightarrow{T} \langle pc \mapsto 3, x \mapsto 2 \rangle$$

(T: operational semantics of program or circuit)

The **Model Checking** problem:

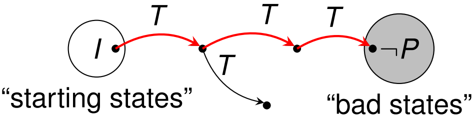


T



(T: operational semantics of program or circuit)

The **Model Checking** problem:



Program Model and Specifications

- Models are finite state (as *Kripke Structure*)
- Specifications are given in *Temporal Logic*

Finite-state transition systems and Kripke structures

Definition

A triple $\langle S, T, I \rangle$ is a *Finite-State Transition System*, if

a *finite* set of states S,

a set of initial states $I \subseteq S$, and

a total transition relation $T \subseteq S \times S$.

(i.e., $\forall s \in S \,.\, \exists s' \in S \,.\, T(s, s')$)

Finite-state transition systems and Kripke structures

Definition

A triple $\langle S, T, I \rangle$ is a *Finite-State Transition System*, if

a *finite* set of states S,

a set of initial states $I \subseteq S$, and

a total transition relation $T \subseteq S \times S$.

(i.e., $\forall s \in S \,.\, \exists s' \in S \,.\, T(s, s')$)

Definition

A quadruple $\langle S, T, I, L \rangle$ is a *Kripke structure*, if

$\langle S, T, I \rangle$ comprise a finite-state transition system,

L is a *labelling function* $S \to 2^{AP}$.

(from the states to a set of atomic propositions AP)

Atomic propositions and assertions

Atomic propositions represent properties of states

(alternatively, we could directly refer to state variables)

Assertion is a Boolean combination of atomic propositions from *AP*

We write $s \models F$ if F holds in state s:

$$
\begin{array}{lcl}
s \models p & \Leftrightarrow & p \in L(s) \\
s \models \neg F & \Leftrightarrow & s \not\models F \\
s \models F_1 \vee F_2 & \Leftrightarrow & s \models F_1 \text{ or } s \models F_2 \\
s \models F_1 \wedge F_2 & \Leftrightarrow & s \models F_1 \text{ and } s \models F_2
\end{array}
$$

Specifying correctness with assertions

Consider a traffic lights control

Each traffic light in the system can be in one of three states:

(In some countries, there are more combinations!)

Specifying Correctness

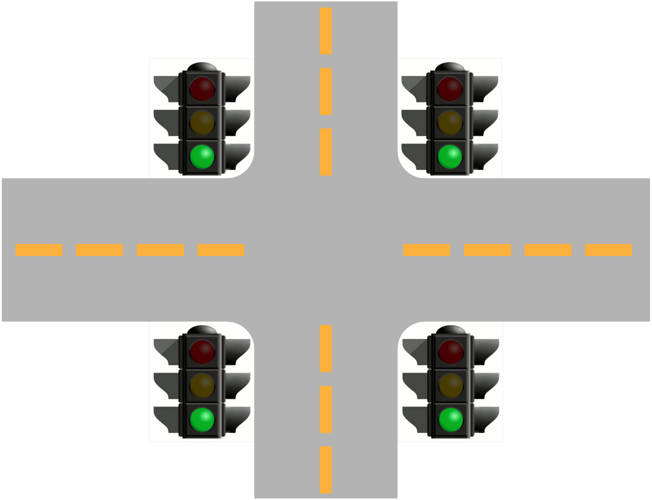
So far, we have specified correctness in terms of assertions

Consider a crossing with two traffic lights $_1$ and $_2$

assert $\Big(\neg$ $_1 \quad \vee \quad \neg$ $_2\Big)$

Enables us to specify "*safety*" of a system

A state of a four-light system



assertion expresses something bad not supposed to happen

The simplest explicit-state model checker

Algorithm EXPLICITREACHDFS

Input:

1. a Kripke structure $\langle S, T, I, L \rangle$,
2. an assertion F

// check, whether every state reachable from I via T satisfies F

```
open := list(I)
visited = ∅
while open ≠ [] {
 s := head(open)
 open := tail(open)
 if s ⊭ F then error(s)
 for each s' ∈ S : (s, s') ∈ T
  if s' ∉ visited then {
     visited := {s'} ∪ visited
     open := s' :: open
  }
}
```

Questions about EXPLICITREACHDFS

1. Why does EXPLICITREACHDFS terminate?
2. How to implement the set operations?
3. How to implement **for each** $s' \in S: (s, s') \in T$ efficiently?
4. How many iterations does the outer loop make (worst case)?
5. How many times is line 8 called (worst case)?
6. Can we report an execution that leads to an error?

The simplest explicit-state model checker (v. 2)

Algorithm EXPLICITREACHDFSCEX
Input:

1. a Kripke structure $\langle S, T, I, L \rangle$,
2. an assertion F

// check, whether every state reachable from I via T satisfies F

// if not, report an execution that leads to a bug

```
visited = ∅
function dfs(s) {
  if s ⊭ F then error(stack())
  for each s′ ∈ S : (s, s′) ∈ T
   if s′ ∉ visited then {
     visited := {s′} ∪ visited
     dfs(s′)
  }
}
for each s ∈ I { dfs(s) }
```

Explicit enumeration with breadth-first search

Algorithm ExplicitReachBFS

Input:

1. a Kripke structure $\langle S, T, I, L \rangle$,
2. an assertion F

// check, whether every state reachable from I via T satisfies F

```
open := list(I)
visited = ∅
while open ≠ [] {
 s := head(open)
 open := tail(open)
 if s ⊭ F then error(s)
 for each s′ ∈ S : (s, s′) ∈ T
  if s′ ∉ visited then {
     visited := {s′} ∪ visited
     open := append(open, s′)
  }
}
```

Depth-first vs. breadth-first

+ BFS always finds a shortest counterexample

- DFS counterexamples can be quite long

+ DFS keeps only the current stack, so $|open| \leq |S|$

- In BFS, *open* tends to grow large (think of duplicate states)

DFS is considered to be more efficient than BFS

SPIN uses DFS, but also supports BFS

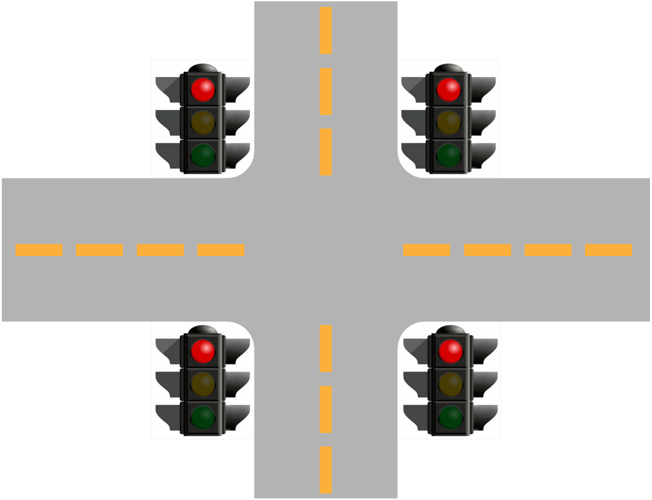
TLC uses BFS

Limits of assertions

What if we want to <u>guarantee</u> that something good happens?

Consider the crossing with two traffic lights  $_1$ and  $_2$

"not indefinitely ( $_1$ $\wedge$  $_2$)"



A perfectly *safe* situation (at least until the drivers lose their temper)

Specifying Correctness

It is impossible to specify this requirement with *assertions*

We have to extend the specification language

Let us revisit the transition systems we are considering

For the time being, we still stick to finite state systems

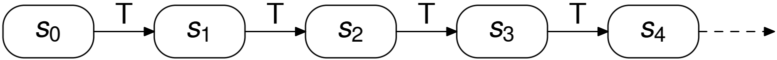
Temporal logics

Paths

Can we reason about paths?

An (infinite) **path** π:

a sequence of states $s_0, s_1, \ldots$ with $T(s_i, s_{i+1})$ for $i \geq 0$



π^i denotes the *suffix* of π starting at s_i

(note that $\pi = \pi^0$)

Path formulas

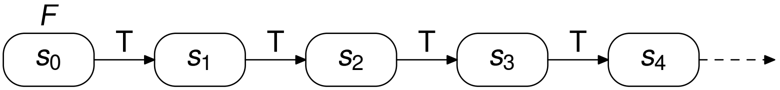
Fix a Kripke structure $\mathcal{M}$ and a path π

We will introduce *path* formulas

...and write $\mathcal{M}, \pi \models \varphi$ to denote that φ holds on the path π

Start with a Boolean combination F of atomic propositions

$$\mathcal{M}, \pi \models F \qquad \Leftrightarrow \qquad ?$$



Path formulas

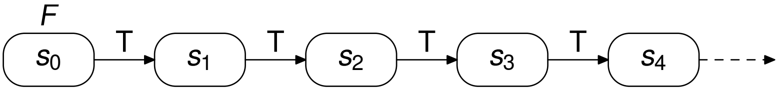
Fix a Kripke structure $\mathcal{M}$ and a path π

We will introduce *path* formulas

...and write $\mathcal{M}, \pi \models \varphi$ to denote that φ holds on the path π

Start with a Boolean combination F of atomic propositions

$$\mathcal{M}, \pi \models F \quad \Leftrightarrow \quad F \text{ holds in first state } s_0 \text{ of } \pi$$



Path formulas

Syntactic convention:

F denotes a *state formula*

φ denotes a *path formula*

We introduce a number of temporal operators,

...which specify what is supposed to happen *along a path*

In what follows, we introduce temporal logic called CTL*

Temporal operators: next

Syntax

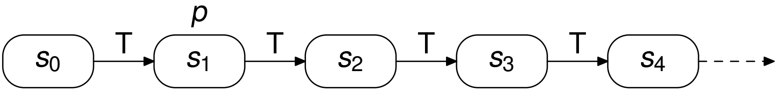
Unary: **X** ⟨path formula⟩

Semantics

$\mathcal{M}, \pi \models \mathbf{X}\,\varphi$ if and only if $\mathcal{M}, \pi^1 \models \varphi$

Example: $\mathcal{M}, \pi \models \mathbf{X}\, p$

(It *doesn't matter* whether or not p holds in s_0 or $s_2, s_3, \ldots$)



Temporal operators: next

Syntax

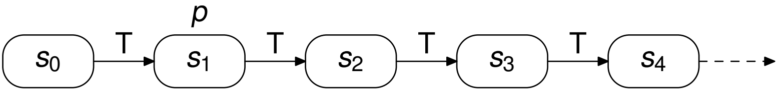
<u>Unary</u>: **X** ⟨path formula⟩

Semantics

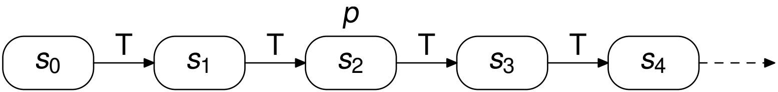
$\mathcal{M}, \pi \models \mathbf{X}\,\varphi$ if and only if $\mathcal{M}, \pi^1 \models \varphi$

Example: $\mathcal{M}, \pi \models \mathbf{X}\,p$

(It *doesn't matter* whether or not p holds in s_0 or $s_2, s_3, \ldots$)



X can be nested: $\mathcal{M}, \pi \models \mathbf{XX}p$



Temporal operators: eventually

Syntax

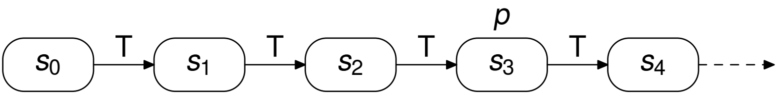
Unary: **F** ⟨path formula⟩

Semantics

$\mathcal{M}, \pi \models \mathbf{F}\,\varphi$ if and only if $\mathcal{M}, \pi^k \models \varphi$ for some $k \geq 0$

Intuitively, *p* holds after a *finite* number of steps

Example: $\mathcal{M}, \pi \models \mathbf{F}p$



F allows us to express basic liveness properties

Temporal operators: globally

Syntax

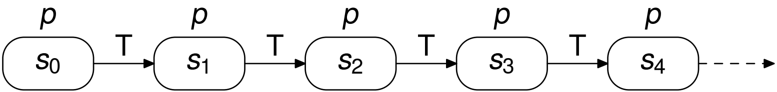
Unary: **G** ⟨path formula⟩

Semantics

$\mathcal{M}, \pi \models \mathbf{G}\,\varphi$ if and only if $\mathcal{M}, \pi^i \models \varphi$ for $i \geq 0$

Intuitively, *p* holds in *every* path state

Example: $\mathcal{M}, \pi \models \mathbf{G}p$



G allows us to express basic safety properties

Temporal operators: until

Syntax

Binary: ⟨path formula⟩ **U** ⟨path formula⟩

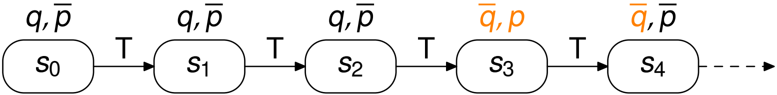
Semantics

$$\mathcal{M}, \pi \models \varphi_1 \mathbf{U} \varphi_2$$

if and only if

there is $k \geq 0$ such that $\mathcal{M}, \pi^k \models \varphi_2$ and $\mathcal{M}, \pi^j \models \varphi_1$ for $0 \leq j < k$

Intuitively, φ_1 holds until φ_2 holds
Importantly, φ_2 must happen eventually!
Example: $\mathcal{M}, \pi \models q \mathbf{U} p$



(q doesn't have to hold anymore once discharged by p)

Temporal operators: release

Syntax

Binary: $\langle$path formula$\rangle$ **R** $\langle$path formula$\rangle$

Semantics

$$\mathcal{M}, \pi \models \varphi_1 \mathbf{R}\, \varphi_2$$

if and only if one of the two conditions holds:

1. $\exists k \geq 0$ such that $\mathcal{M}, \pi^k \models \varphi_1$ and $\mathcal{M}, \pi^j \models \varphi_2$ for $0 \leq j < k$
2. $\mathcal{M}, \pi^j \models \varphi_2$ for $j \geq 0$

Temporal operators: release

Syntax

Binary: ⟨path formula⟩ **R** ⟨path formula⟩

Semantics

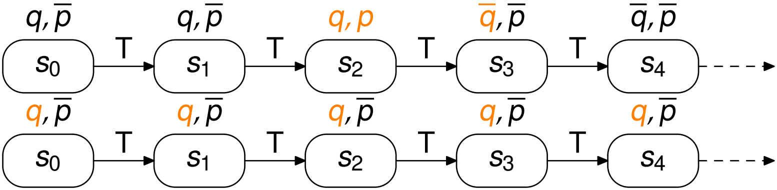
$$\mathcal{M}, \pi \models \varphi_1 \, \mathbf{R} \, \varphi_2$$

if and only if one of the two conditions holds:

1. $\exists k \geq 0$ such that $\mathcal{M}, \pi^k \models \varphi_1$ and $\mathcal{M}, \pi^j \models \varphi_2$ for $0 \leq j < k$
2. $\mathcal{M}, \pi^j \models \varphi_2$ for $j \geq 0$

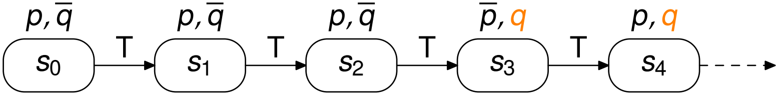
φ_1 releases φ_2 (if φ_2 ever holds)

Example: $\mathcal{M}, \pi \models p \, \mathbf{R} \, q$



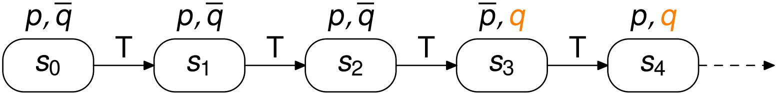
Temporal operators: more examples

$$\mathcal{M}, \pi \models p\,\mathbf{U}\,(\mathbf{G}\,q)$$

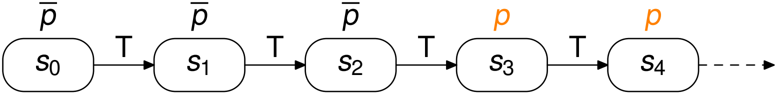


Temporal operators: more examples

$$\mathcal{M}, \pi \models p\,\mathbf{U}\,(\mathbf{G}\,q)$$



$$\mathcal{M}, \pi \models \mathbf{F}\,(\mathbf{G}\,p)$$



Temporal operators: more examples

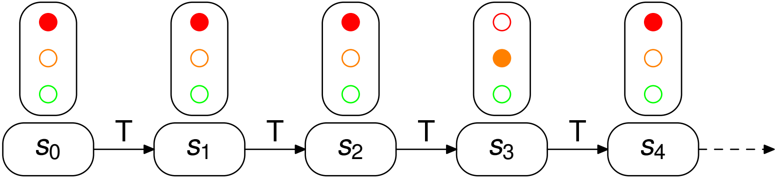
"not indefinitely ()"

$\mathcal{M}, \pi \models$ **F** $\left(\neg\right.$  $\left.\right)$ or $\mathcal{M}, \pi \models \neg$**G** $\left(\right.$  $\left.\right)$

Temporal operators: more examples

"not indefinitely ()"

$\mathcal{M}, \pi \models \mathbf{F}\left(\neg\right.$  $\left.\right)$ or $\mathcal{M}, \pi \models \neg\mathbf{G}\left(\right.$  $\left.\right)$



Temporal operators: equivalences

As the last example shows,

...some temporal operators can be rewritten in terms of others:

$$\mathbf{G}\,\varphi \equiv \neg\mathbf{F}\,(\neg\varphi)$$

$$\mathbf{F}\,\varphi \equiv \text{true}\,\mathbf{U}\,\varphi$$

$$\varphi_1\,\mathbf{R}\,\varphi_2 \equiv \neg(\neg\varphi_1\,\mathbf{U}\,\neg\varphi_2)$$

$\neg$, **X**, **U** are sufficient to express **G**, **F**, and **R**

(c.f. “basis” ($\neg, \vee$) in propositional logic)

Temporal operators: path quantifiers

So far, we could only talk about individual paths

To amend this, we introduce *path quantifiers*

Syntax

E $\langle$path formula$\rangle$

A $\langle$path formula$\rangle$

Semantics

$\mathcal{M}, s \models \mathbf{E}\varphi \quad \Leftrightarrow \quad \exists\pi$ starting at s such that $\mathcal{M}, \pi \models \varphi$

$\mathcal{M}, s \models \mathbf{A}\varphi \quad \Leftrightarrow \quad \forall\pi$ starting at s it holds that $\mathcal{M}, \pi \models \varphi$

Note that $\mathbf{E}\varphi$ and $\mathbf{A}\varphi$ are state formulas!

Remember:

unwinding a Kripke structure results in infinite tree

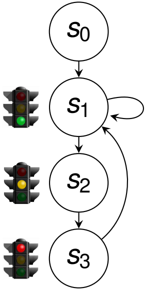
The introduced logic is called

Computation Tree Logic*

(or just CTL*)

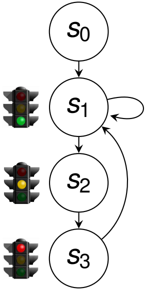
* As you probably have guessed, there is also CTL, discussed later

Computation Tree Logic (CTL*): Examples



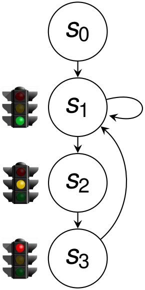
$\mathcal{M}, s_0 \models \mathbf{AF}$ ()

Computation Tree Logic (CTL*): Examples



$\mathcal{M}, s_0 \models \mathbf{AF}$ () ✓

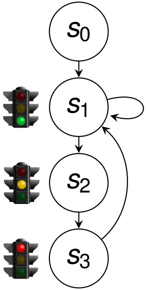
Computation Tree Logic (CTL*): Examples



$\mathcal{M}, s_0 \models$ **AF** () ✓

$\mathcal{M}, s_0 \models$ **AX** (**EG** ())

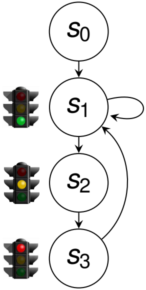
Computation Tree Logic (CTL*): Examples



$\mathcal{M}, s_0 \models \mathbf{AF}$ () ✓

$\mathcal{M}, s_0 \models \mathbf{AX}$ ($\mathbf{EG}$ ()) ✓

Computation Tree Logic (CTL*): Examples

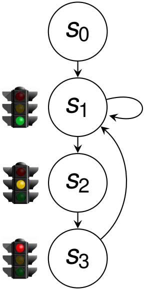


$\mathcal{M}, s_0 \models$ **AF** () ✓

$\mathcal{M}, s_0 \models$ **EGX** ()

$\mathcal{M}, s_0 \models$ **AX** (**EG** ()) ✓

Computation Tree Logic (CTL*): Examples

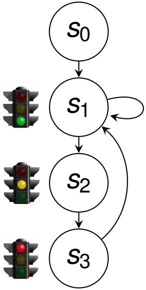


$\mathcal{M}, s_0 \models$ **AF** () ✓

$\mathcal{M}, s_0 \models$ **EGX** () ✓

$\mathcal{M}, s_0 \models$ **AX** (**EG** ()) ✓

Computation Tree Logic (CTL*): Examples



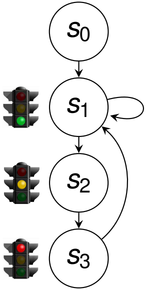
$\mathcal{M}, s_0 \models$ **AF** () ✓

$\mathcal{M}, s_0 \models$ **EGX** () ✓

$\mathcal{M}, s_0 \models$ **AX** (**EG** ()) ✓

$\mathcal{M}, s_0 \models$ **AGX** ()

Computation Tree Logic (CTL*): Examples



$\mathcal{M}, s_0 \models$ **AF** () ✓

$\mathcal{M}, s_0 \models$ **EGX** () ✓

$\mathcal{M}, s_0 \models$ **AX** (**EG** ()) ✓

$\mathcal{M}, s_0 \models$ **AGX** () ×

Branching Time vs. Linear Time

Commonly used fragments of CTL*:

branching-time logic

quantifies over paths possible from a given state

linear-time logic

for events along a single computation path only

Branching Time: Computation Tree Logic (CTL)

CTL restricts CTL* formulas:

X, **F**, **G**, **U**, and **R** must be immediately preceded by **A** or **E**

Examples:

EF (*start* $\wedge \neg$*ready*)	there's a path on which we start at some point despite not being ready
AG(*req*$\Rightarrow$ **AF** *ack*)	each request eventually acknowledged
AG EX *progress*	no deadlocks

every CTL formula is also a CTL* formula (by construction)

Linear Temporal Logic (LTL)

Linear Temporal Logic also restricts CTL* (differently than CTL)

A CTL* formula is an LTL formula, if there is a formula ψ:

(a) φ starts with **A**, that is, $\varphi \equiv \mathbf{A}\psi$

(b) ψ contains neither **E**, nor **A**

intuitively, φ is interpreted over all paths

every LTL formula is also a CTL* formula (by construction)

Wondering, whether you could use **E** instead of **A**?

That would be the logic called ELTL

LTL: examples

$\mathbf{A}(\mathbf{FG}\, p)$ "all paths eventually stabilise with property p"

(cannot be expressed in CTL)

$\mathbf{A}(\mathbf{GF}\, p)$ "p is visited infinitely often"

$\mathbf{AG}(\mathit{try} \rightarrow \mathbf{F}\, \text{succeed})$ "every attempt eventually succeeds"

Bored to write **A** in front of a formula? We too! Usually, **A** is omitted

Preview: The SPIN Explicit State Model Checker

`http://spinroot.com`

- “Explicit-state” Model Checker
- Models with asynchronous processes
- Communication via channels
  - Modeling language PROMELA

LTL in the SPIN Model Checker

- Unary Operators

`[]`	Globally ($\Box$ or `G`)
`<>`	Eventually ($\Diamond$ or `F`)
`!`	Boolean negation

- Binary Operators

`U`	Until
`&&`	Boolean "and"
`||`	Boolean "or"
`->`	Boolean Implication

Temporal Properties for PROMELA Traffic Light

```
ltl P1 { [] <> g1 }
ltl P2 { [] ! (g1 && g2) }

active proctype TrafficLight2() {
  do
  :: an1 -> g1 = 1
  :: aus1 -> g1 = 0
  od
}
active proctype TrafficLight2() {
  do
  :: an2 -> g2 = 1
  :: aus2 -> g2 = 0
  od
}
active proctype Control() {
  do
  :: c == 1 -> an1 = 1; aus1 = 0; c = 2;
  :: c == 2 -> an1 = 0; aus1 = 1; c = 3;
  :: c == 3 -> an2 = 1; aus2 = 0; c = 4;
  :: c == 4 -> an2 = 0; aus2 = 1; c = 1;
  od
}
```

Model Checking of PROMELA Models with SPIN

- Generate C program from PROMELA:

```
spin -a traffic.pml
```

- Compile program (gcc necessary):

```
gcc -o pan -DBFS pan.c
```

- Start model checking:

```
./pan -N P2 traffic.pml
```

Model Checking of PROMELA Models with SPIN

- Generate C program from PROMELA:

```
spin -a traffic.pml
```

- Compile program (gcc necessary):

```
gcc -o pan -DBFS pan.c
```

- Start model checking:

```
./pan -N P2 traffic.pml
```

- Result:

```
assertion violated !( !( !((g1&&g2))))
```

Model Checking of PROMELA Models with SPIN

- Generate C program from PROMELA:

```
spin -a traffic.pml
```

- Compile program (gcc necessary):

```
gcc -o pan -DBFS pan.c
```

- Start model checking:

```
./pan -N P2 traffic.pml
```

- Result:

```
assertion violated !( !( !((g1&&g2))))
```

- View counterexample:

```
./pan -r traffic.pml.trail
```

Expressiveness CTL*, CTL, and LTL

A CTL* formula φ *distinguishes* logic *A* from logic *B*, if:

(a) φ is a formula of *A*, and

(b) no formula of *B* is equivalent to φ

(equivalent formulas are satisfied by the same Kripke structures)

$\mathbf{A\,F\,G}\ p$ and $\mathbf{AF}(p \wedge \mathbf{X}\ p)$ distinguish LTL from CTL

$\mathbf{AG\ EF}\ p$ and $\mathbf{AFAG}\ p$ distinguish CTL from LTL

want more? $\mathbf{AF}(p \wedge \mathbf{AX}\ p)$ distinguishes CTL from LTL too

$(\mathbf{A\,F\,G}p) \vee (\mathbf{AG\ EF}\ p)$ distinguishes CTL* from CTL and LTL

Proofs: Baier, Katoen (2008), pp. 337 and 424

Complexity of CTL*, CTL, and LTL

Consider a Kripke structure $\langle S, T, I, L \rangle$ and a CTL* formula φ

- $|S|$ and $|T|$ are the number of states and transitions resp.
- $|\varphi|$ is the number of φ's subformulas

Table: Complexity of model checking for fragments of CTL*

CTL	LTL	CTL*
PTIME	PSPACE-complete	PSPACE-complete
$O(\|\varphi\| \cdot (\|S\| + \|T\|))$	$O(2^{\|\varphi\|} \cdot (\|S\| + \|T\|))$	$O(2^{\|\varphi\|} \cdot (\|S\| + \|T\|))$

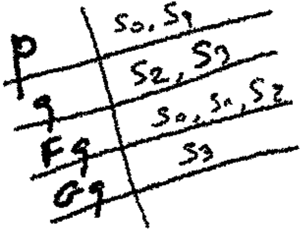
Details: Baier, Katoen (2008), pp. 430

Good news: we consider only the algorithm for CTL

(explicit)

tableaux model checking

for CTL



Model Checking for CTL

Fix a finite Kripke structure $M = \langle S, T, I, L \rangle$

Notation: $[\![\psi]\!] \stackrel{\text{def}}{=} \{s \in S \mid M, s \models \psi\}$ for a CTL formula ψ

CTL model checking problem:

for a CTL formula φ, answer, whether $I \subseteq [\![\varphi]\!]$

Thus, our goal is to compute the set $[\![\varphi]\!]$

Preprocessing step: simplify formulas

CTL has 10 basic operators

	X	**F**	**G**	**U**	**R**
A	**AX**	**AF**	**AG**	**AU**	**AR**
E	**EX**	**EF**	**EG**	**EU**	**ER**

all 10 can be expressed in terms of **EX**, **EG**, and **EU**:

$$\mathbf{AX}\varphi \equiv \neg\mathbf{EX}(\neg\varphi) \qquad \mathbf{EF}\varphi \equiv \mathbf{E}(\text{true}\,\mathbf{U}\,\varphi)$$

$$\mathbf{AG}\varphi \equiv \neg\mathbf{EF}(\neg\varphi) \qquad \mathbf{AF}\varphi \equiv \neg\mathbf{EG}(\neg\varphi)$$

$$\mathbf{A}(\varphi_1\,\mathbf{R}\,\varphi_2) \equiv \neg\mathbf{E}(\neg\varphi_1\,\mathbf{U}\,\neg\varphi_2) \qquad \mathbf{E}(\varphi_1\,\mathbf{R}\,\varphi_2) \equiv \neg\mathbf{A}(\neg\varphi_1\,\mathbf{U}\,\neg\varphi_2)$$

$$\mathbf{A}(\varphi_1\,\mathbf{U}\,\varphi_2) \equiv \neg\mathbf{E}(\neg\varphi_2\,\mathbf{U}\,(\neg\varphi_1 \wedge \neg\varphi_2)) \wedge \neg\mathbf{EG}\neg\varphi_2$$

Tableaux structure

Using syntactic structure of φ (parse tree), construct the set $\mathcal{T}_\varphi$

Start with $\mathcal{T}_\varphi = \{\varphi\}$, apply the rules until no applicable rule left:

1. if $\psi' \wedge \psi'' \in \mathcal{T}_\varphi$, then $\mathcal{T}_\varphi := \{\psi', \psi''\} \cup \mathcal{T}_\varphi$
2. if $\neg\psi \in \mathcal{T}_\varphi$, then $\mathcal{T}_\varphi := \{\psi\} \cup \mathcal{T}_\varphi$
3. if **EX** $\psi \in \mathcal{T}_\varphi$, then $\mathcal{T}_\varphi := \{\psi\} \cup \mathcal{T}_\varphi$
4. if **EG** $\psi \in \mathcal{T}_\varphi$, then $\mathcal{T}_\varphi := \{\psi\} \cup \mathcal{T}_\varphi$
5. if ψ' **EU** $\psi'' \in \mathcal{T}_\varphi$, then $\mathcal{T}_\varphi := \{\psi', \psi''\} \cup \mathcal{T}_\varphi$

Tableaux structure

Using syntactic structure of φ (parse tree), construct the set $\mathcal{T}_\varphi$

Start with $\mathcal{T}_\varphi = \{\varphi\}$, apply the rules until no applicable rule left:

1. if $\psi' \wedge \psi'' \in \mathcal{T}_\varphi$, then $\mathcal{T}_\varphi := \{\psi', \psi''\} \cup \mathcal{T}_\varphi$
2. if $\neg\psi \in \mathcal{T}_\varphi$, then $\mathcal{T}_\varphi := \{\psi\} \cup \mathcal{T}_\varphi$
3. if $\textbf{EX}\,\psi \in \mathcal{T}_\varphi$, then $\mathcal{T}_\varphi := \{\psi\} \cup \mathcal{T}_\varphi$
4. if $\textbf{EG}\,\psi \in \mathcal{T}_\varphi$, then $\mathcal{T}_\varphi := \{\psi\} \cup \mathcal{T}_\varphi$
5. if $\psi'\,\textbf{EU}\,\psi'' \in \mathcal{T}_\varphi$, then $\mathcal{T}_\varphi := \{\psi', \psi''\} \cup \mathcal{T}_\varphi$

Example: for $\varphi \equiv (\textbf{EX}\,\textbf{EF}\,p) \wedge \textbf{EG}\,q$,

we have $\mathcal{T}_\varphi = \{\varphi, \textbf{EX}\,\textbf{EF}\,p, \textbf{EF}\,p, p, \textbf{EG}\,q, q\}$.

Tableaux computation

Having constructed the set $\mathcal{T}_\varphi$,
we will compute $[\![\psi]\!]$ for each $\psi \in \mathcal{T}_\varphi$

We start from the bottom (propositions) and end at the top (φ)

Tableaux computation

Having constructed the set $\mathcal{T}_\varphi$,

we will compute $[\![\psi]\!]$ for each $\psi \in \mathcal{T}_\varphi$

We start from the bottom (propositions) and end at the top (φ)

In our example, $\varphi \equiv (\mathbf{EX}\,\mathbf{EF}\,p) \wedge \mathbf{EG}\,q$

Our goal is to fill the table:

$(\mathbf{EX}\,\mathbf{EF}\,p) \wedge \mathbf{EG}\,q$	?
$\mathbf{EX}\,\mathbf{EF}\,p$	?
$\mathbf{EF}\,p$	?
$\mathbf{EG}\,q$	?
p	?
q	?

Easy part: propositions and Boolean connectives

Propositions are very easy to handle:

$[\![p]\!] = \{s \in S \mid p \in L(s)\}$ for $p \in AP$

Booleans are easy too:

$[\![\psi' \wedge \psi'']\!] = [\![\psi']\!] \cap [\![\psi'']\!]$ and $[\![\neg\psi]\!] = S \setminus [\![\psi]\!]$

Easy part: propositions and Boolean connectives

Propositions are very easy to handle:

$[\![p]\!] = \{s \in S \mid p \in L(s)\}$ for $p \in AP$

Booleans are easy too:

$[\![\psi' \wedge \psi'']\!] = [\![\psi']\!] \cap [\![\psi'']\!]$ and $[\![\neg\psi]\!] = S \setminus [\![\psi]\!]$

$(\mathbf{EX}\,\mathbf{EF}\,p) \wedge \mathbf{EG}\,q$	$[\![\mathbf{EX}\,\mathbf{EF}\,p]\!] \cap [\![\mathbf{EG}\,q]\!]$
$\mathbf{EX}\,\mathbf{EF}\,p$	?
$\mathbf{EF}\,p$	?
$\mathbf{EG}\,q$	?
p	$\{s \in S \mid p \in L(s)\}$
q	$\{s \in S \mid q \in L(s)\}$

Nexttime

The first *really* temporal operator is easy too:

```
procedure compEX(ψ) {
    [[EXψ]] := {s ∈ S | ∃s' ∈ [[ψ]] and (s, s') ∈ T}
}
```

Note the relation between **E** and $\exists$ and between **X** and T

Until

φ **EU** ψ (or **E** φ **U** ψ) requires us to reason about paths:

```
procedure compEU(ψ′, ψ″) {
   Z  := ∅
   Z′  := ⟦ψ″⟧
   while  Z ≠ Z′
      Z  :=  Z′
      Z′  :=  Z ∪ {s ∈ ⟦ψ′⟧ | ∃s′ ∈ Z  and  (s, s′) ∈ T}

   ⟦ψ′ EU ψ″⟧ := Z
}
```

Why does it terminate?

Globally

EG ψ requires us to find cycles on which ψ always holds

- We start with $[\![\psi]\!]$
- Then we *eliminate* states *s* with no successor in $[\![\psi]\!]$

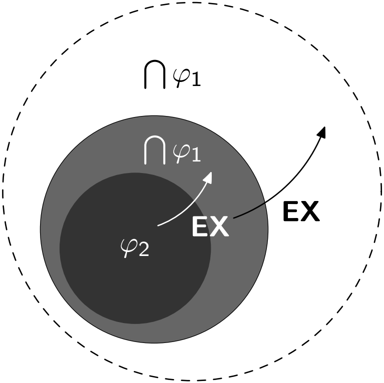
```
procedure compEG(ψ) {
   Z' := [[ψ]]
   do
      Z := Z'
      Z' := {s ∈ [[ψ]] | ∃s' ∈ Z and (s, s') ∈ T}
   while Z ≠ Z'

   [[EG ψ]] := Z
 }
```

Complete algorithm

Algorithm ExplicitCTL
Input: a Kripke structure $M = \langle S, T, I, L \rangle$ and a CTL formula φ

```
compute 𝒯_φ = {ψ_0, ..., ψ_k} such that |ψ_0| ≥ ··· ≥ |ψ_k|
for i from k downto 0 {
 if ψ_i = p such that p ∈ AP then
   ⟦ψ_i⟧ := {s ∈ S | p ∈ L(s)}
 if ψ_i = ¬ψ then
   ⟦ψ_i⟧ := S \ ⟦ψ⟧
 if ψ_i = ψ′ ∧ ψ″ then
   ⟦ψ_i⟧ := ⟦ψ′⟧ ∩ ⟦ψ″⟧
 if ψ_i = EX ψ then
   ⟦ψ_i⟧ := compEX(ψ)
 if ψ_i = EG ψ then
   ⟦ψ_i⟧ := compEG(ψ)
 if ψ_i = ψ′ EU ψ″ then
   ⟦ψ_i⟧ := compEU(ψ′, ψ″)
}
```

Illustration of EU

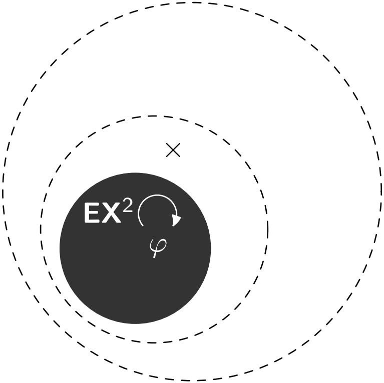


$\mathbf{E}(\varphi_1\,\mathbf{U}\,\varphi_2)$ holds in φ_2

and in predecessor states of φ_2 in which φ_1 holds

Fixed point: Transitive closure of all such predecessor states

Illustration of EG

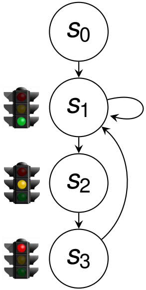


Start with all states in which φ holds

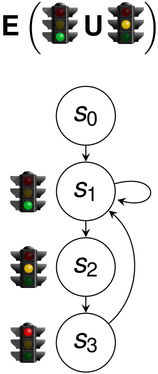
shrink to states in φ such that φ still holds after 1 step

Keep shrinking until fixed point reached

Traffic light and EU



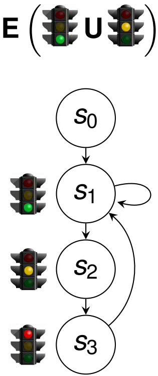
Traffic light and EU

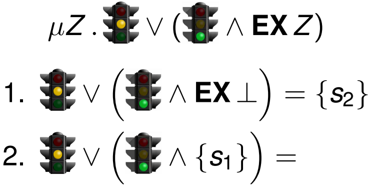


μZ .  $\vee$ ( $\wedge$ **EX** Z)

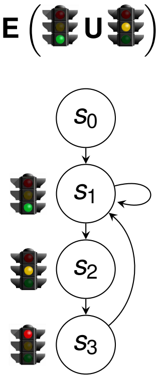
1.  $\vee$ ( $\wedge$ **EX** $\bot$) $= \{s_2\}$
2.  $\vee$ ( $\wedge$ **EX** $\{s_2\}$) $=$

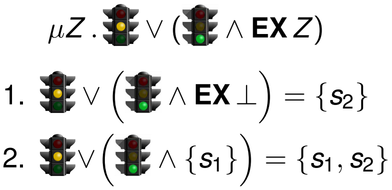
Traffic light and EU



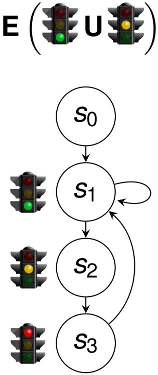


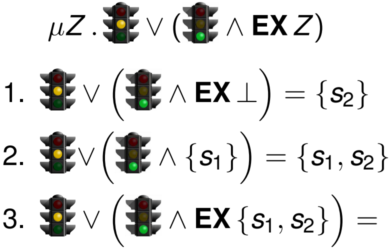
Traffic light and EU



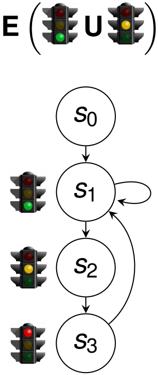


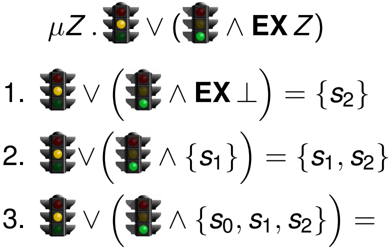
Traffic light and EU



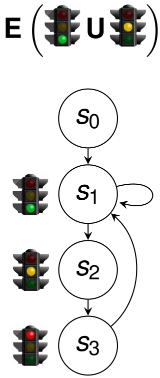


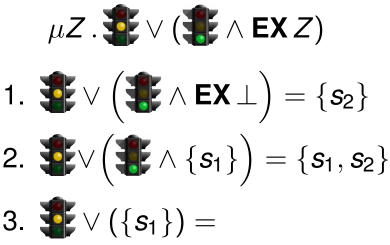
Traffic light and EU



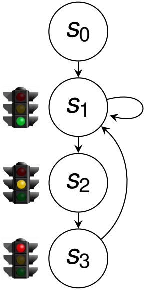


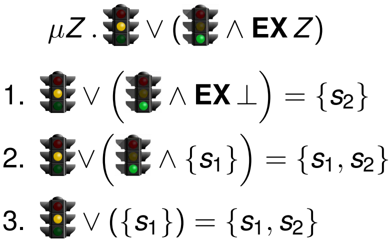
Traffic light and EU





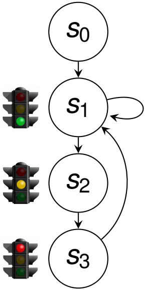
Traffic light and EU





Traffic light and EU

$\mathbf{E}\,($🚦green $\mathbf{U}$ 🚦yellow$)$



$\mu Z\,.$ 🚦yellow $\vee\,($🚦green $\wedge\ \mathbf{EX}\ Z)$

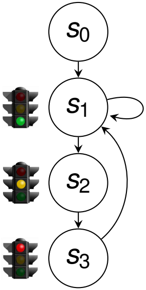
1. 🚦yellow $\vee\ ($🚦green $\wedge\ \mathbf{EX}\ \bot) = \{s_2\}$
2. 🚦yellow $\vee\ ($🚦green $\wedge\ \{s_1\}) = \{s_1, s_2\}$
3. 🚦yellow $\vee\ (\{s_1\}) = \{s_1, s_2\}$
4. Fixed point!

- $\mathcal{M}, s_1 \models \mathbf{E}\,($🚦green $\mathbf{U}$ 🚦yellow$)$
- $\mathcal{M}, s_2 \models \mathbf{E}\,($🚦green $\mathbf{U}$ 🚦yellow$)$

Traffic light and EG

- Let's compute the greatest fixed point

$$\nu Z \,.\, \{s_1, s_2\} \wedge \mathbf{EX}\, Z$$

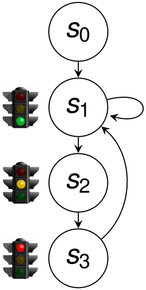


1. $\{s_1, s_2\} \wedge \mathbf{EX}\, \top$

Traffic light and EG

- Let's compute the greatest fixed point

$$\nu Z \, . \, \{s_1, s_2\} \wedge \textbf{EX}\, Z$$

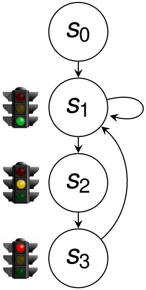


1. $\{s_1, s_2\} \wedge \top$

Traffic light and EG

- Let's compute the greatest fixed point

$$\nu Z \,.\, \{s_1, s_2\} \wedge \textbf{EX}\, Z$$

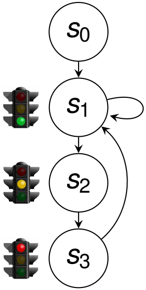


1. $\{s_1, s_2\} \wedge \top = \{s_1, s_2\}$

Traffic light and EG

- Let's compute the greatest fixed point

$$\nu Z \,.\, \{s_1, s_2\} \wedge \mathbf{EX}\, Z$$

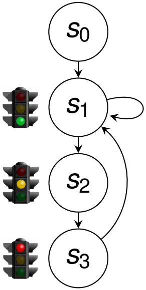


1. $\{s_1, s_2\} \wedge \top = \{s_1, s_2\}$
2. $\{s_1, s_2\} \wedge \mathbf{EX}\,\{s_1, s_2\}$

Traffic light and EG

- Let's compute the greatest fixed point

$$\nu Z \,.\, \{s_1, s_2\} \wedge \mathbf{EX}\, Z$$

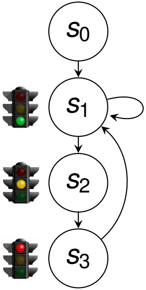


1. $\{s_1, s_2\} \wedge \top = \{s_1, s_2\}$
2. $\{s_1, s_2\} \wedge \{s_0, s_1, s_3\}$

Traffic light and EG

- Let's compute the greatest fixed point

$$\nu Z \,.\, \{s_1, s_2\} \wedge \mathbf{EX}\, Z$$

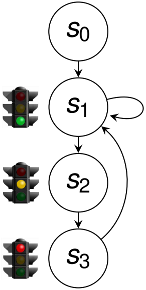


1. $\{s_1, s_2\} \wedge \top = \{s_1, s_2\}$
2. $\{s_1, s_2\} \wedge \{s_0, s_1, s_3\} = \{s_1\}$

Traffic light and EG

- Let's compute the greatest fixed point

$$\nu Z \,.\, \{s_1, s_2\} \wedge \mathbf{EX}\, Z$$

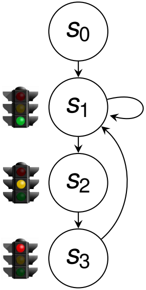


1. $\{s_1, s_2\} \wedge \top = \{s_1, s_2\}$
2. $\{s_1, s_2\} \wedge \{s_0, s_1, s_3\} = \{s_1\}$
3. $\{s_1, s_2\} \wedge \mathbf{EX}\, \{s_1\}$

Traffic light and EG

- Let's compute the greatest fixed point

$$\nu Z \,.\, \{s_1, s_2\} \wedge \mathbf{EX}\, Z$$

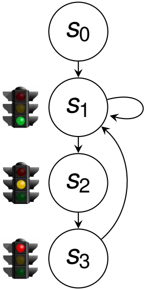


1. $\{s_1, s_2\} \wedge \top = \{s_1, s_2\}$
2. $\{s_1, s_2\} \wedge \{s_0, s_1, s_3\} = \{s_1\}$
3. $\{s_1, s_2\} \wedge \{s_0, s_1, s_3\}$

Traffic light and EG

- Let's compute the greatest fixed point

$$\nu Z\,.\,\{s_1, s_2\} \wedge \mathbf{EX}\,Z$$

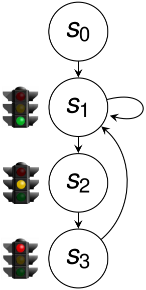


1. $\{s_1, s_2\} \wedge \top = \{s_1, s_2\}$
2. $\{s_1, s_2\} \wedge \{s_0, s_1, s_3\} = \{s_1\}$
3. $\{s_1, s_2\} \wedge \{s_0, s_1, s_3\} = \{s_1\}$

Traffic light and EG

- Let's compute the greatest fixed point

$$\nu Z \,.\, \{s_1, s_2\} \wedge \mathbf{EX}\, Z$$



1. $\{s_1, s_2\} \wedge \top = \{s_1, s_2\}$
2. $\{s_1, s_2\} \wedge \{s_0, s_1, s_3\} = \{s_1\}$
3. $\{s_1, s_2\} \wedge \{s_0, s_1, s_3\} = \{s_1\}$
4. Fixed point!

$\mathcal{M}, s_1 \models \mathbf{EG}\Big(\mathbf{E}\big($  $\mathbf{U}$  $\big)\Big)$

Complexity?

CompEX requires $O(|T|)$

CompEF and CompEG require $O(|S| + |T|)$ operations

Propositions, $\neg$, and $\wedge$ can be treated in $O(|S|)$

Thus, $O(|\varphi| \cdot (|S| + |T|))$

Summary

- Introduced *temporal logics* as a specification language
  - Branching time logic (CTL)
  - Linear time logic (LTL)
  - Computation tree logic (CTL^*)
- Explicit model checking for CTL