Automated Program Verification and Testing 15414/15614 Fall 2016 Lecture 20: Explicit-State Model Checking, Part 1

Matt Fredrikson
mfredrik@cs.cmu.edu

November 15, 2016

Useful Temporal Properties (Review)

Typically, the properties we want to check fall into a few categories

Useful Temporal Properties (Review)

Typically, the properties we want to check fall into a few categories

- **Safety:** Properties which require that “nothing bad should ever happen”

Useful Temporal Properties (Review)

Typically, the properties we want to check fall into a few categories

- **Safety:** Properties which require that "nothing bad should ever happen"
- **Liveness:** Require that "something good always happens" in the future

Useful Temporal Properties (Review)

Typically, the properties we want to check fall into a few categories

- **Safety:** Properties which require that "nothing bad should ever happen"
- **Liveness:** Require that "something good always happens" in the future
- **Fairness:** Precludes "degenerate" infinite behaviors

Useful Temporal Properties (Review)

Typically, the properties we want to check fall into a few categories

- **Safety:** Properties which require that "nothing bad should ever happen"
- **Liveness:** Require that "something good always happens" in the future
- **Fairness:** Precludes "degenerate" infinite behaviors

These are inherently linear-time properties

Safety: Invariants

An important class of safety properties describes **invariant** behavior

Safety: Invariants

An important class of safety properties describes **invariant** behavior

Invariants state that some property over states holds at all times

Safety: Invariants

An important class of safety properties describes **invariant** behavior

Invariants state that some property over states holds at all times

- Mutual exclusion

Safety: Invariants

An important class of safety properties describes **invariant** behavior

Invariants state that some property over states holds at all times

- Mutual exclusion
- Deadlock freedom

Safety: Invariants

An important class of safety properties describes **invariant** behavior

Invariants state that some property over states holds at all times

- Mutual exclusion
- Deadlock freedom
- Well-formedness of data structures

Safety: Invariants

An important class of safety properties describes **invariant** behavior

Invariants state that some property over states holds at all times

- Mutual exclusion
- Deadlock freedom
- Well-formedness of data structures

If ϕ is a propositional formula over P, then the LTL formula

$$\mathbf{G}\ \phi$$

is an invariant property

Example: Mutual Exclusion (Review)

Consider a concurrent system with processes P_1 and P_2

Example: Mutual Exclusion (Review)

Consider a concurrent system with processes P_1 and P_2

What is ϕ?

Example: Mutual Exclusion (Review)

Consider a concurrent system with processes P_1 and P_2

What is ϕ? P_1 and P_2 can't be simultaneously critical

Example: Mutual Exclusion (Review)

Consider a concurrent system with processes P_1 and P_2

What is ϕ? P_1 and P_2 can't be simultaneously critical

$$\phi \Leftrightarrow \neg c_1 \vee \neg c_2$$

Example: Mutual Exclusion (Review)

Consider a concurrent system with processes P_1 and P_2

What is ϕ? P_1 and P_2 can't be simultaneously critical

$$\phi \Leftrightarrow \neg c_1 \vee \neg c_2$$

Algorithmically, invariant checking is a reachability problem

Example: Mutual Exclusion (Review)

Consider a concurrent system with processes P_1 and P_2

What is ϕ? P_1 and P_2 can't be simultaneously critical

$$\phi \Leftrightarrow \neg c_1 \vee \neg c_2$$

Algorithmically, invariant checking is a reachability problem

To see if **G** ϕ holds, check for a state $\neg\phi$ reachable from I

Safety properties, beyond invariants (Review)

Violating an invariant is one type of “bad thing” that could happen

Safety properties, beyond invariants (Review)

Violating an invariant is one type of “bad thing” that could happen

Generally, we’re interested in more than simple reachability

Safety properties, beyond invariants (Review)

Violating an invariant is one type of “bad thing” that could happen

Generally, we’re interested in more than simple reachability

General safety properties are defined in terms of **bad path prefixes**

Safety properties, beyond invariants (Review)

Violating an invariant is one type of “bad thing” that could happen

Generally, we’re interested in more than simple reachability

General safety properties are defined in terms of **bad path prefixes**

- Let ϕ be a property over paths

Safety properties, beyond invariants (Review)

Violating an invariant is one type of “bad thing” that could happen

Generally, we’re interested in more than simple reachability

General safety properties are defined in terms of **bad path prefixes**

- Let ϕ be a property over paths
- Let $\hat{\sigma}$ be a prefix such that no path $\sigma = \hat{\sigma} \cdots$ satisfies ϕ

Safety properties, beyond invariants (Review)

Violating an invariant is one type of "bad thing" that could happen

Generally, we're interested in more than simple reachability

General safety properties are defined in terms of **bad path prefixes**

- Let ϕ be a property over paths
- Let $\hat{\sigma}$ be a prefix such that no path $\sigma = \hat{\sigma} \cdots$ satisfies ϕ
- Note: we require $\hat{\sigma}$ to be finite

Safety properties, beyond invariants (Review)

Violating an invariant is one type of "bad thing" that could happen

Generally, we're interested in more than simple reachability

General safety properties are defined in terms of **bad path prefixes**

- Let ϕ be a property over paths
- Let $\hat{\sigma}$ be a prefix such that no path $\sigma = \hat{\sigma} \cdots$ satisfies ϕ
- Note: we require $\hat{\sigma}$ to be finite

ϕ is a safety property iff every path that violates ϕ has a bad prefix

Safety properties, beyond invariants (Review)

Violating an invariant is one type of “bad thing” that could happen

Generally, we’re interested in more than simple reachability

General safety properties are defined in terms of **bad path prefixes**

- Let ϕ be a property over paths
- Let $\hat{\sigma}$ be a prefix such that no path $\sigma = \hat{\sigma} \cdots$ satisfies ϕ
- Note: we require $\hat{\sigma}$ to be finite

ϕ is a safety property iff every path that violates ϕ has a bad prefix

Think of a bad prefix as a witness of a violating instance

Liveness Properties (Review)

Safety properties prevent bad things from happening

Liveness Properties (Review)

Safety properties prevent bad things from happening

It's easy to design systems that don't do bad things

Liveness Properties (Review)

Safety properties prevent bad things from happening

It's easy to design systems that don't do bad things: **do nothing**

Liveness Properties (Review)

Safety properties prevent bad things from happening

It's easy to design systems that don't do bad things: **do nothing**

Usually this isn't what we want, and **liveness** properties address this

Liveness Properties (Review)

Safety properties prevent bad things from happening

It's easy to design systems that don't do bad things: **do nothing**

Usually this isn't what we want, and **liveness** properties address this

Intuitively, "something good" will always happen in the future

Liveness Properties (Review)

Safety properties prevent bad things from happening

It's easy to design systems that don't do bad things: **do nothing**

Usually this isn't what we want, and **liveness** properties address this

Intuitively, "something good" will always happen in the future

This intuition reveals a key distinction from safety properties:

Liveness Properties (Review)

Safety properties prevent bad things from happening

It's easy to design systems that don't do bad things: **do nothing**

Usually this isn't what we want, and **liveness** properties address this

Intuitively, "something good" will always happen in the future

This intuition reveals a key distinction from safety properties:

- Safety properties are violated in **finite time**

Liveness Properties (Review)

Safety properties prevent bad things from happening

It's easy to design systems that don't do bad things: **do nothing**

Usually this isn't what we want, and **liveness** properties address this

Intuitively, "something good" will always happen in the future

This intuition reveals a key distinction from safety properties:

- Safety properties are violated in **finite time**
- Liveness properties can only be violated in **infinite time**

Liveness, Formally

Liveness, Formally

Liveness properties are defined in terms of **infinite path extensions**

Liveness, Formally

Liveness properties are defined in terms of **infinite path extensions**

- Let ϕ be a property over paths

Liveness, Formally

Liveness properties are defined in terms of **infinite path extensions**

- Let ϕ be a property over paths
- Let $\hat{\sigma}$ be a finite path prefix

Liveness, Formally

Liveness properties are defined in terms of **infinite path extensions**

- Let ϕ be a property over paths
- Let $\hat{\sigma}$ be a finite path prefix
- σ is an infinite path extension if $\hat{\sigma}\sigma$ satisfies ϕ

Liveness, Formally

Liveness properties are defined in terms of **infinite path extensions**

- Let ϕ be a property over paths
- Let $\hat{\sigma}$ be a finite path prefix
- σ is an infinite path extension if $\hat{\sigma}\sigma$ satisfies ϕ

ϕ is a liveness property iff every finite prefix has an infinite extension

Liveness, Formally

Liveness properties are defined in terms of **infinite path extensions**

- Let ϕ be a property over paths
- Let $\hat{\sigma}$ be a finite path prefix
- σ is an infinite path extension if $\hat{\sigma}\sigma$ satisfies ϕ

ϕ is a liveness property iff every finite prefix has an infinite extension

Put differently: a liveness property does not rule out any finite prefix

Liveness, Formally

Liveness properties are defined in terms of **infinite path extensions**

- Let ϕ be a property over paths
- Let $\hat{\sigma}$ be a finite path prefix
- σ is an infinite path extension if $\hat{\sigma}\sigma$ satisfies ϕ

ϕ is a liveness property iff every finite prefix has an infinite extension

Put differently: a liveness property does not rule out any finite prefix

This implies that there are no finite witnesses for liveness

Liveness: Examples

Think of the mutual exclusion example

Liveness: Examples

Think of the mutual exclusion example

Desirable liveness property: each P_i enters *critical*$_i$ infinitely often

Liveness: Examples

Think of the mutual exclusion example

Desirable liveness property: each P_i enters *critical*$_i$ infinitely often

$$\textbf{G}\ \textbf{F}\ c_1 \land \textbf{G}\ \textbf{F}\ c_2$$

Liveness: Examples

Think of the mutual exclusion example

Desirable liveness property: each P_i enters *critical*$_i$ infinitely often

$$\mathbf{G}\ \mathbf{F}\ c_1 \wedge \mathbf{G}\ \mathbf{F}\ c_2$$

Another: each waiting process eventually enters its *critical*$_i$

Liveness: Examples

Think of the mutual exclusion example

Desirable liveness property: each P_i enters *critical*$_i$ infinitely often

$$\mathbf{G}\ \mathbf{F}\ c_1 \wedge \mathbf{G}\ \mathbf{F}\ c_2$$

Another: each waiting process eventually enters its *critical*$_i$

$$\mathbf{G}\ (w_i \rightarrow \mathbf{F}\ c_i)$$

Liveness: Examples

Think of the mutual exclusion example

Desirable liveness property: each P_i enters *critical*$_i$ infinitely often

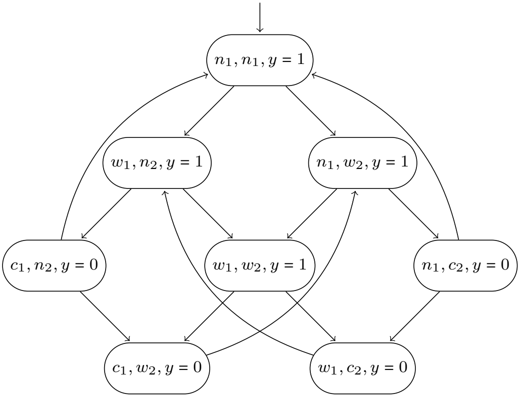
$$\mathbf{G}\ \mathbf{F}\ c_1 \wedge \mathbf{G}\ \mathbf{F}\ c_2$$

Another: each waiting process eventually enters its *critical*$_i$

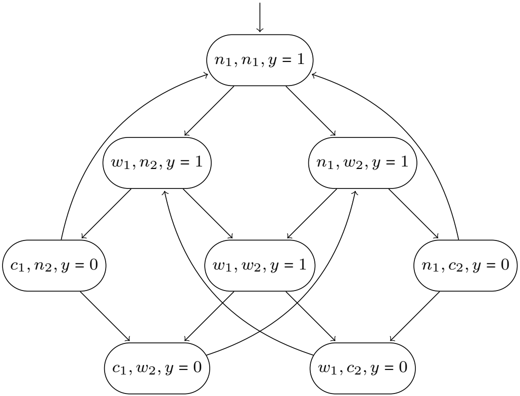
$$\mathbf{G}\ (w_i \rightarrow \mathbf{F}\ c_i)$$

Any finite prefix can always be extended to satisfy these properties

Liveness: “Repeated Eventually”

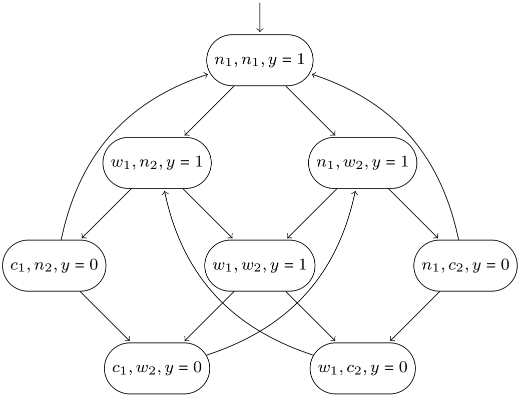


Liveness: "Repeated Eventually"



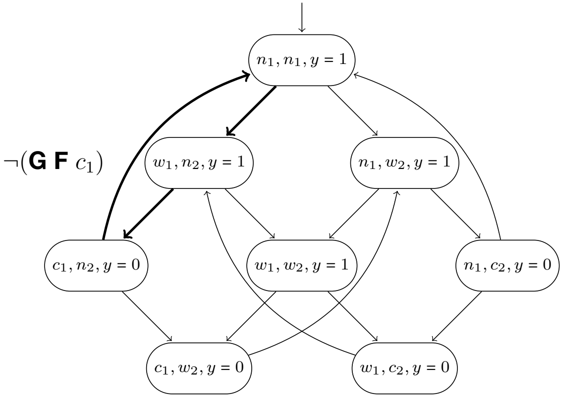
Does this satisfy **G F** $c_1 \wedge$ **G F** c_2?

Liveness: "Repeated Eventually"



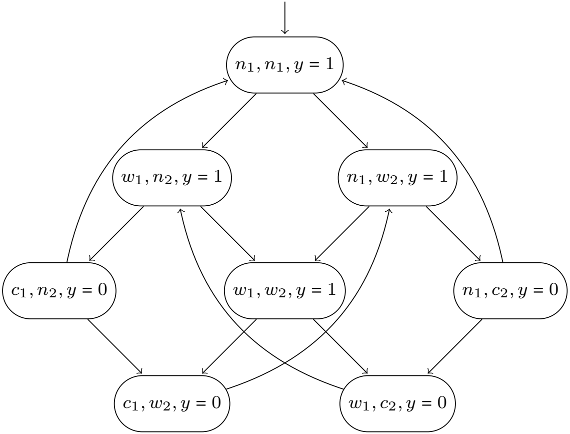
Does this satisfy **G F** $c_1 \wedge$ **G F** c_2? **No.** What's a counterexample?

Liveness: “Repeated Eventually”

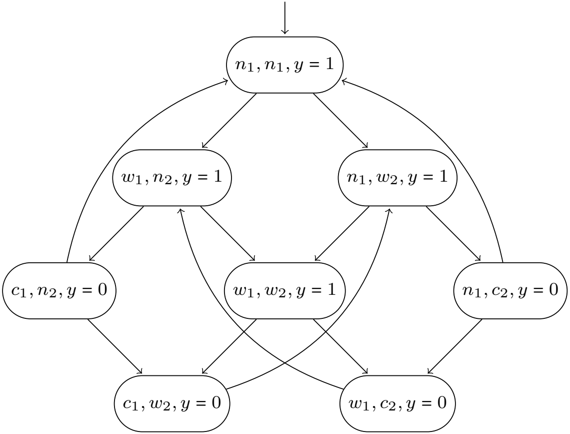


Does this satisfy $\mathbf{G}\ \mathbf{F}\ c_1 \wedge \mathbf{G}\ \mathbf{F}\ c_2$? **No.** What’s a counterexample?

Liveness: Starvation Freedom

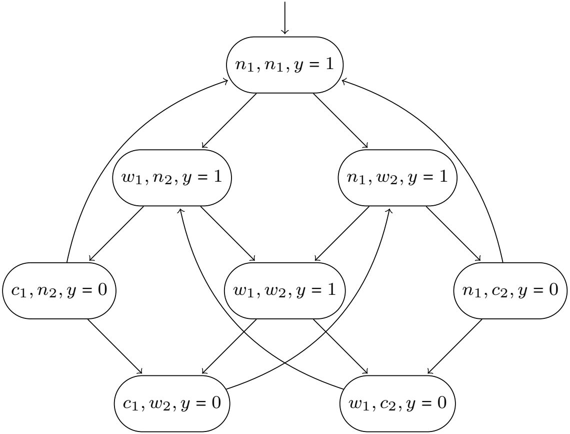


Liveness: Starvation Freedom



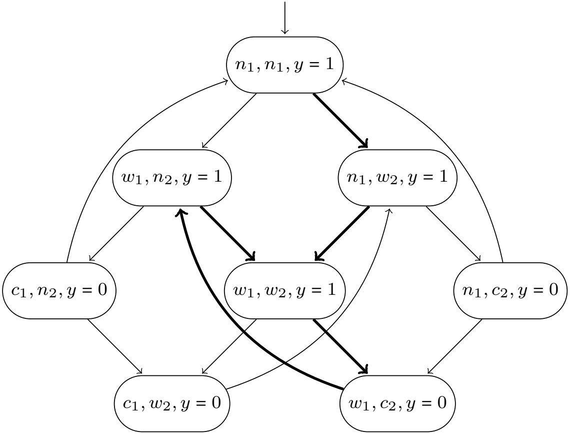
Does this satisfy **G** $(w_i \rightarrow$ **F** $c_i)$?

Liveness: Starvation Freedom



Does this satisfy **G** $(w_i \rightarrow$ **F** $c_i)$? **No.** What's a counterexample?

Liveness: Starvation Freedom



Does this satisfy **G** $(w_i \to$ **F** $c_i)$? **No.** What's a counterexample?

Fairness

Liveness violated because processes aren't scheduled **fairly**

Fairness

Liveness violated because processes aren't scheduled **fairly**

Unfairness arises from nondeterminism

Fairness

Liveness violated because processes aren't scheduled **fairly**

Unfairness arises from nondeterminism

- Choices that are globally biased against certain options

Fairness

Liveness violated because processes aren't scheduled **fairly**

Unfairness arises from nondeterminism

- Choices that are globally biased against certain options
- Often these choices are unrealistic in practice

Fairness

Liveness violated because processes aren't scheduled **fairly**

Unfairness arises from nondeterminism

- Choices that are globally biased against certain options
- Often these choices are unrealistic in practice

To prove liveness, we want to rule out unfair paths from the model

Fairness

Liveness violated because processes aren't scheduled **fairly**

Unfairness arises from nondeterminism

- Choices that are globally biased against certain options
- Often these choices are unrealistic in practice

To prove liveness, we want to rule out unfair paths from the model

We accomplish this by ruling out unfair paths when checking liveness

Fairness

Liveness violated because processes aren't scheduled **fairly**

Unfairness arises from nondeterminism

- Choices that are globally biased against certain options
- Often these choices are unrealistic in practice

To prove liveness, we want to rule out unfair paths from the model

We accomplish this by ruling out unfair paths when checking liveness

Fairness constraints allow us to do this

Fairness Constraints

There are several different types of fairness

Fairness Constraints

There are several different types of fairness

- **Unconditional fairness**: Every process is executed infinitely often.

$$\text{for all } i, \textbf{G F } P_i$$

Fairness Constraints

There are several different types of fairness

- **Unconditional fairness**: Every process is executed infinitely often.

$$\text{for all } i, \textbf{G F } P_i$$

- **Strong fairness**: Every process that is enabled infinitely often gets its turn infinitely often when it is enabled

$$\text{for all } i, (\textbf{G F } \textit{enabled}_i) \rightarrow (\textbf{G F } P_i)$$

Fairness Constraints

There are several different types of fairness

- **Unconditional fairness**: Every process is executed infinitely often.

  $$\text{for all } i, \textbf{G F } P_i$$

- **Strong fairness**: Every process that is enabled infinitely often gets its turn infinitely often when it is enabled

  $$\text{for all } i, (\textbf{G F } \textit{enabled}_i) \rightarrow (\textbf{G F } P_i)$$

- **Weak fairness**: Every process that is at some point continuously enabled gets its turn infinitely often

  $$\text{for all } i, (\textbf{F G } \textit{enabled}_i) \rightarrow (\textbf{G F } P_i)$$

Fairness Constraints

There are several different types of fairness

- **Unconditional fairness**: Every process is executed infinitely often.

  $$\text{for all } i, \mathbf{G}\ \mathbf{F}\ P_i$$

- **Strong fairness**: Every process that is enabled infinitely often gets its turn infinitely often when it is enabled

  $$\text{for all } i, (\mathbf{G}\ \mathbf{F}\ \textit{enabled}_i) \rightarrow (\mathbf{G}\ \mathbf{F}\ P_i)$$

- **Weak fairness**: Every process that is at some point continuously enabled gets its turn infinitely often

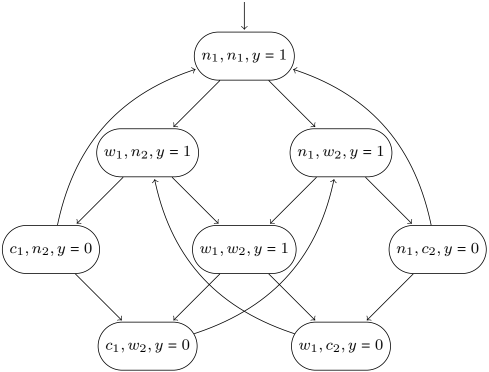
  $$\text{for all } i, (\mathbf{F}\ \mathbf{G}\ \textit{enabled}_i) \rightarrow (\mathbf{G}\ \mathbf{F}\ P_i)$$

"Enabled" means "able to execute" a particular transition

Example: Weak Fairness

Consider the mutual exclusion example

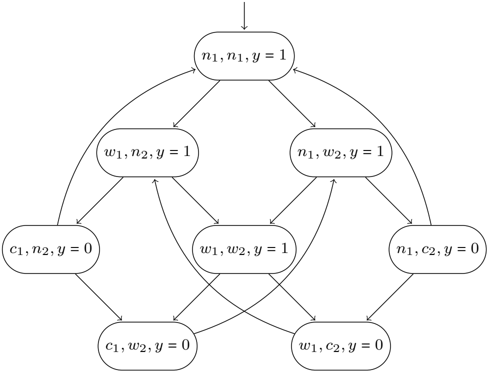
A process is **enabled** if able to enter its critical section immediately



Example: Weak Fairness

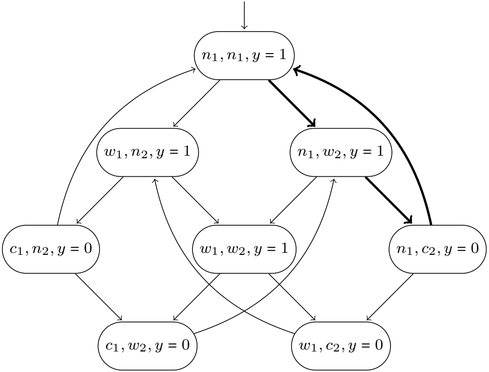
Consider the mutual exclusion example

A process is **enabled** if able to enter its critical section immediately



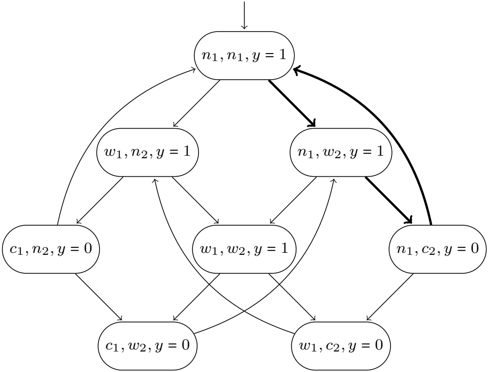
Does this satisfy **G** $(w_i \rightarrow$ **F** $c_i)$ under weak fairness?

Example: Weak Fairness



No, **G** $(w_i \to$ **F** $c_i)$ isn't satisfied under weak fairness

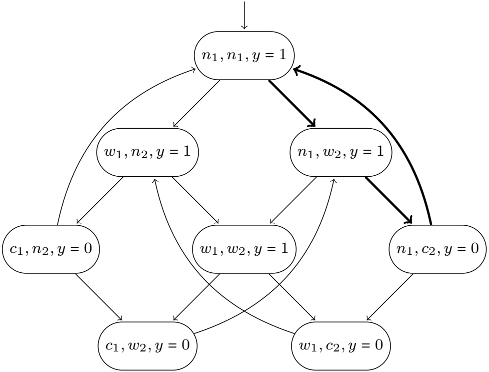
Example: Weak Fairness



No, **G** $(w_i \to$ **F** $c_i)$ isn't satisfied under weak fairness

Whenever P_2 enters *critical*, P_1 is not enabled

Example: Weak Fairness

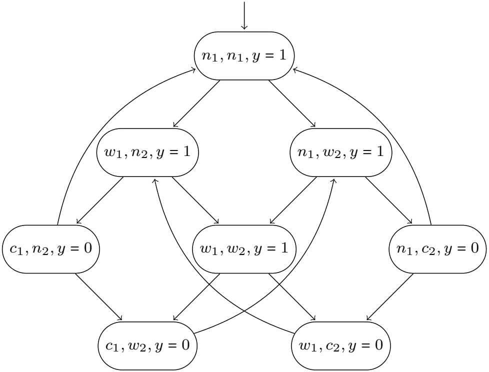


No, **G** $(w_i \rightarrow$ **F** $c_i)$ isn't satisfied under weak fairness

Whenever P_2 enters *critical*, P_1 is not enabled

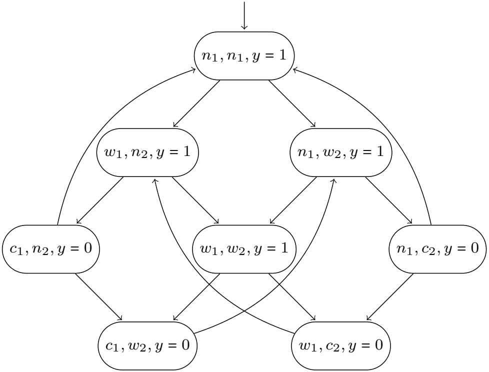
The antecedent that P_i be enabled continuously is false

Example: Strong Fairness



Does this satisfy **G** $(w_i \to$ **F** $c_i)$ under strong fairness?

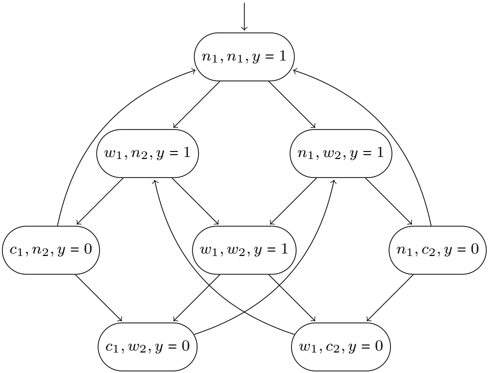
Example: Strong Fairness



Does this satisfy **G** $(w_i \to$ **F** $c_i)$ under strong fairness?

Yes, both processes enabled infinitely often along all paths

Example: Strong Fairness



Does this satisfy **G** $(w_i \rightarrow$ **F** $c_i)$ under strong fairness?

Yes, both processes enabled infinitely often along all paths

So a strongly-fair scheduler lets both execute infinitely often

Example: Unconditional Fairness

Does starvation freedom hold if the scheduler is unconditionally fair?

Unconditional Fairness : for all i, **G F** P_i

Example: Unconditional Fairness

Does starvation freedom hold if the scheduler is unconditionally fair?

Unconditional Fairness : for all i, **G F** P_i

Yes. It held for strong fairness, which was:

Strong Fairness : for all i, (**G F** *enabled*$_i$) $\rightarrow$ (**G F** P_i)

Example: Unconditional Fairness

Does starvation freedom hold if the scheduler is unconditionally fair?

Unconditional Fairness : for all i, **G F** P_i

Yes. It held for strong fairness, which was:

Strong Fairness : for all i, $($**G F** *enabled*$_i) \rightarrow ($**G F** $P_i)$

So we see that unconditional fairness implies strong fairness

Example: Unconditional Fairness

Does starvation freedom hold if the scheduler is unconditionally fair?

Unconditional Fairness : for all i, **G F** P_i

Yes. It held for strong fairness, which was:

Strong Fairness : for all i, (**G F** *enabled*$_i$) $\rightarrow$ (**G F** P_i)

So we see that unconditional fairness implies strong fairness

In fact, we can say that:

Unconditional Fairness $\Longrightarrow$ Strong Fairness $\Longrightarrow$ Weak Fairness

Checking CTL

CTL Model Checking Problem

Given a transition system $M = (P, S, I, L, R)$ and a CTL formula ϕ, check whether:

$$M, s \models \phi \text{ for all } s \in I$$

Checking CTL

CTL Model Checking Problem

Given a transition system $M = (P, S, I, L, R)$ and a CTL formula ϕ, check whether:

$$M, s \models \phi \text{ for all } s \in I$$

We'll take the following approach:

Checking CTL

CTL Model Checking Problem

Given a transition system $M = (P, S, I, L, R)$ and a CTL formula ϕ, check whether:

$$M, s \models \phi \text{ for all } s \in I$$

We'll take the following approach:

1. Compute *Sat*(ϕ), the set of all states that satisfy ϕ

Checking CTL

CTL Model Checking Problem

Given a transition system $M = (P, S, I, L, R)$ and a CTL formula ϕ, check whether:

$$M, s \models \phi \text{ for all } s \in I$$

We'll take the following approach:

1. Compute *Sat*(ϕ), the set of all states that satisfy ϕ
2. Check to make sure that $I \subseteq$ *Sat*(ϕ)

Checking CTL

CTL Model Checking Problem

Given a transition system $M = (P, S, I, L, R)$ and a CTL formula ϕ, check whether:

$$M, s \models \phi \text{ for all } s \in I$$

We'll take the following approach:

1. Compute *Sat*(ϕ), the set of all states that satisfy ϕ
2. Check to make sure that $I \subseteq$ *Sat*(ϕ)

This is stronger than model checking: know all states that satisfy ϕ

Checking CTL

CTL Model Checking Problem

Given a transition system $M = (P, S, I, L, R)$ and a CTL formula ϕ, check whether:

$$M, s \models \phi \text{ for all } s \in I$$

We'll take the following approach:

1. Compute *Sat*(ϕ), the set of all states that satisfy ϕ
2. Check to make sure that $I \subseteq$ *Sat*(ϕ)

This is stronger than model checking: know all states that satisfy ϕ

Sometimes called "global" model checking

Satisfiable Set: $Sat(\phi)$

To reduce the number of cases, we'll assume a normal form

Satisfiable Set: *Sat*(ϕ)

To reduce the number of cases, we'll assume a normal form

Existential Normal Form (ENF)

$$\phi ::= \textit{true} \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \textbf{EX}\ \phi \mid \textbf{E}\ (\phi_1\ \textbf{U}\ \phi_2) \mid \textbf{EG}\ \phi$$

Satisfiable Set: *Sat*(ϕ)

To reduce the number of cases, we'll assume a normal form

Existential Normal Form (ENF)

$$\phi ::= \textit{true} \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \textbf{EX}\ \phi \mid \textbf{E}\ (\phi_1\ \textbf{U}\ \phi_2) \mid \textbf{EG}\ \phi$$

Exercise: Convince yourself we can rewrite formulas in ENF

Satisfiable Set: *Sat*(ϕ)

To reduce the number of cases, we'll assume a normal form

Existential Normal Form (ENF)

$$\phi ::= \textit{true} \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \textbf{EX}\ \phi \mid \textbf{E}\ (\phi_1\ \textbf{U}\ \phi_2) \mid \textbf{EG}\ \phi$$

Exercise: Convince yourself we can rewrite formulas in ENF

Given ENF ϕ and $M = (P, S, I, L, R)$, we define *Sat*(ϕ) as follows:

Satisfiable Set: *Sat*(ϕ)

To reduce the number of cases, we'll assume a normal form

Existential Normal Form (ENF)

$$\phi ::= \textit{true} \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \textbf{EX}\ \phi \mid \textbf{E}\ (\phi_1\ \textbf{U}\ \phi_2) \mid \textbf{EG}\ \phi$$

Exercise: Convince yourself we can rewrite formulas in ENF

Given ENF ϕ and $M = (P, S, I, L, R)$, we define *Sat*(ϕ) as follows:

- *Sat*(*true*) = S

Satisfiable Set: *Sat*(ϕ)

To reduce the number of cases, we'll assume a normal form

Existential Normal Form (ENF)

$$\phi ::= \textit{true} \mid \phi_1 \land \phi_2 \mid \neg\phi \mid \textbf{EX}\ \phi \mid \textbf{E}\ (\phi_1\ \textbf{U}\ \phi_2) \mid \textbf{EG}\ \phi$$

Exercise: Convince yourself we can rewrite formulas in ENF

Given ENF ϕ and $M = (P, S, I, L, R)$, we define *Sat*(ϕ) as follows:

- *Sat*(*true*) = S
- *Sat*(p) = $\{s \in S | p \in L(s)\}$

Satisfiable Set: *Sat*(ϕ)

To reduce the number of cases, we'll assume a normal form

Existential Normal Form (ENF)

$$\phi ::= \textit{true} \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \textbf{EX}\ \phi \mid \textbf{E}\ (\phi_1\ \textbf{U}\ \phi_2) \mid \textbf{EG}\ \phi$$

Exercise: Convince yourself we can rewrite formulas in ENF

Given ENF ϕ and $M = (P, S, I, L, R)$, we define *Sat*(ϕ) as follows:

- *Sat*$(\textit{true}) = S$
- *Sat*$(p) = \{s \in S | p \in L(s)\}$
- *Sat*$(\neg\phi) = S \setminus$ *Sat*(ϕ)

Satisfiable Set: *Sat*(ϕ)

To reduce the number of cases, we'll assume a normal form

Existential Normal Form (ENF)

$$\phi ::= \textit{true} \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \textbf{EX}\ \phi \mid \textbf{E}\ (\phi_1\ \textbf{U}\ \phi_2) \mid \textbf{EG}\ \phi$$

Exercise: Convince yourself we can rewrite formulas in ENF

Given ENF ϕ and $M = (P, S, I, L, R)$, we define *Sat*(ϕ) as follows:

- *Sat*(*true*) = S
- *Sat*(p) = $\{s \in S | p \in L(s)\}$
- *Sat*$(\neg\phi)$ = $S \setminus$ *Sat*(ϕ)
- *Sat*$(\phi \wedge \psi)$ = *Sat*$(\phi) \cap (\psi)$

Satisfiable Set: *Sat*(**EX** ϕ)

For the next rules, we'll refer to direct predecessors and successors:

$$\textbf{Post}(s) = \{s' \in S \mid (s, s') \in R\}$$
$$\textbf{Pre}(s) = \{s' \in S \mid (s', s) \in R\}$$

Satisfiable Set: *Sat*(**EX** ϕ)

For the next rules, we'll refer to direct predecessors and successors:

$$\begin{aligned} \textbf{Post}(s) &= \{s' \in S \mid (s, s') \in R\} \\ \textbf{Pre}(s) &= \{s' \in S \mid (s', s) \in R\} \end{aligned}$$

Continuing on:

$$\mathit{Sat}(\textbf{EX}\ \phi) = \{s \in S \mid \textbf{Post}(s) \cap \mathit{Sat}(\phi) \neq \varnothing\}$$

Satisfiable Set: *Sat*(**EX** ϕ)

For the next rules, we'll refer to direct predecessors and successors:

$$\textbf{Post}(s) = \{s' \in S \mid (s, s') \in R\}$$
$$\textbf{Pre}(s) = \{s' \in S \mid (s', s) \in R\}$$

Continuing on:

$$\textit{Sat}(\textbf{EX}\ \phi) = \{s \in S \mid \textbf{Post}(s) \cap \textit{Sat}(\phi) \neq \varnothing\}$$

In other words, the set of states satisfying **EX** ϕ are:

Satisfiable Set: *Sat*(**EX** ϕ)

For the next rules, we'll refer to direct predecessors and successors:

$$\textbf{Post}(s) = \{s' \in S \mid (s, s') \in R\}$$
$$\textbf{Pre}(s) = \{s' \in S \mid (s', s) \in R\}$$

Continuing on:

$$\textit{Sat}(\textbf{EX}\ \phi) = \{s \in S \mid \textbf{Post}(s) \cap \textit{Sat}(\phi) \neq \varnothing\}$$

In other words, the set of states satisfying **EX** ϕ are:

1. States with at least one direct successor s,

Satisfiable Set: *Sat*(**EX** ϕ)

For the next rules, we'll refer to direct predecessors and successors:

$$\textbf{Post}(s) = \{s' \in S \mid (s, s') \in R\}$$
$$\textbf{Pre}(s) = \{s' \in S \mid (s', s) \in R\}$$

Continuing on:

$$\textit{Sat}(\textbf{EX}\ \phi) = \{s \in S \mid \textbf{Post}(s) \cap \textit{Sat}(\phi) \neq \varnothing\}$$

In other words, the set of states satisfying **EX** ϕ are:

1. States with at least one direct successor s,
2. Where s is in the states that satisfy ϕ

Satisfiable Set: *Sat*(**EG** ϕ)

Now for "exists always"

Sat(**EG** ϕ) is the **largest** subset $T \subseteq S$ where:

Satisfiable Set: *Sat*(**EG** ϕ)

Now for "exists always"

Sat(**EG** ϕ) is the **largest** subset $T \subseteq S$ where:

1. $T \subseteq$ *Sat*(ϕ)

Satisfiable Set: *Sat*(**EG** ϕ)

Now for "exists always"

Sat(**EG** ϕ) is the **largest** subset $T \subseteq S$ where:

1. $T \subseteq$ *Sat*(ϕ)
2. $s \in T$ implies that **Post**$(s) \cap T \neq \varnothing$

Satisfiable Set: *Sat*(**EG** ϕ)

Now for "exists always"

Sat(**EG** ϕ) is the **largest** subset $T \subseteq S$ where:

1. $T \subseteq$ *Sat*(ϕ)
2. $s \in T$ implies that **Post**$(s) \cap T \neq \varnothing$

The set of states satisfying **EG** ϕ are the largest set of:

Satisfiable Set: *Sat*(**EG** ϕ)

Now for "exists always"

Sat(**EG** ϕ) is the **largest** subset $T \subseteq S$ where:

1. $T \subseteq$ *Sat*(ϕ)
2. $s \in T$ implies that **Post**$(s) \cap T \neq \varnothing$

The set of states satisfying **EG** ϕ are the largest set of:

1. States that satisfy ϕ, and

Now for “exists always”

Sat(**EG** ϕ) is the **largest** subset $T \subseteq S$ where:

1. $T \subseteq$ *Sat*(ϕ)
2. $s \in T$ implies that **Post**$(s) \cap T \neq \varnothing$

The set of states satisfying **EG** ϕ are the largest set of:

1. States that satisfy ϕ, and
2. have at least one successor that satisfies **EG** ϕ

Satisfiable Set: *Sat*(**EG** ϕ)

Now for "exists always"

Sat(**EG** ϕ) is the **largest** subset $T \subseteq S$ where:

1. $T \subseteq$ *Sat*(ϕ)
2. $s \in T$ implies that **Post**$(s) \cap T \neq \varnothing$

The set of states satisfying **EG** ϕ are the largest set of:

1. States that satisfy ϕ, and
2. have at least one successor that satisfies **EG** ϕ

Note the recursion: *Sat*(**EG** ϕ) is the largest set T satisfying:

$$T = \{s \in \textit{Sat}(\phi) \mid \textbf{Post}(s) \cap T \neq \varnothing\}$$

Satisfiable Set: *Sat*(**EG** ϕ)

Now for "exists always"

Sat(**EG** ϕ) is the **largest** subset $T \subseteq S$ where:

1. $T \subseteq$ *Sat*(ϕ)
2. $s \in T$ implies that **Post**$(s) \cap T \neq \varnothing$

The set of states satisfying **EG** ϕ are the largest set of:

1. States that satisfy ϕ, and
2. have at least one successor that satisfies **EG** ϕ

Note the recursion: *Sat*(**EG** ϕ) is the largest set T satisfying:

$$T = \{s \in \mathit{Sat}(\phi) \mid \mathbf{Post}(s) \cap T \neq \varnothing\}$$

We say T is a **greatest fixed point solution** of the above equation

Satisfiable Set $Sat(\mathbf{E}\ (\phi\ \mathbf{U}\ \psi))$

$Sat(\mathbf{E}\ (\phi\ \mathbf{U}\ \psi))$ is the **smallest** subset $T \subseteq S$ where:

Satisfiable Set *Sat*($\mathbf{E}$ (ϕ $\mathbf{U}$ ψ))

Sat($\mathbf{E}$ (ϕ $\mathbf{U}$ ψ)) is the **smallest** subset $T \subseteq S$ where:

1. $T \subseteq$ *Sat*(ψ)

Satisfiable Set *Sat*($\mathbf{E}$ (ϕ $\mathbf{U}$ ψ))

Sat($\mathbf{E}$ (ϕ $\mathbf{U}$ ψ)) is the **smallest** subset $T \subseteq S$ where:

1. $T \subseteq$ *Sat*(ψ)
2. $s \in$ *Sat*(ϕ) and **Post**$(s) \cap T \neq \varnothing$ implies that $s \in T$

Satisfiable Set *Sat*($\mathbf{E}$ $(\phi$ $\mathbf{U}$ $\psi))$

Sat($\mathbf{E}$ $(\phi$ $\mathbf{U}$ $\psi))$ is the **smallest** subset $T \subseteq S$ where:

1. $T \subseteq$ *Sat*(ψ)
2. $s \in$ *Sat*(ϕ) and **Post**$(s) \cap T \neq \varnothing$ implies that $s \in T$

We can give a similar fixed-point characterization here:

$$T = \mathit{Sat}(\psi) \cup \{s \in \mathit{Sat}(\phi) \mid \mathbf{Post}(s) \cap T \neq \varnothing\}$$

Satisfiable Set *Sat*(**E** (ϕ **U** ψ))

Sat(**E** (ϕ **U** ψ)) is the **smallest** subset $T \subseteq S$ where:

1. $T \subseteq$ *Sat*(ψ)
2. $s \in$ *Sat*(ϕ) and **Post**$(s) \cap T \neq \varnothing$ implies that $s \in T$

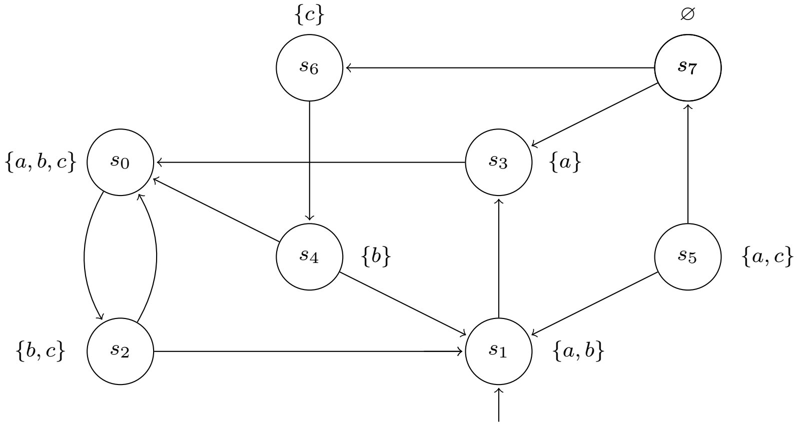
We can give a similar fixed-point characterization here:

$$T = \mathit{Sat}(\psi) \cup \{s \in \mathit{Sat}(\phi) \mid \mathbf{Post}(s) \cap T \neq \varnothing\}$$

In this case, T is a **least fixed point solution**

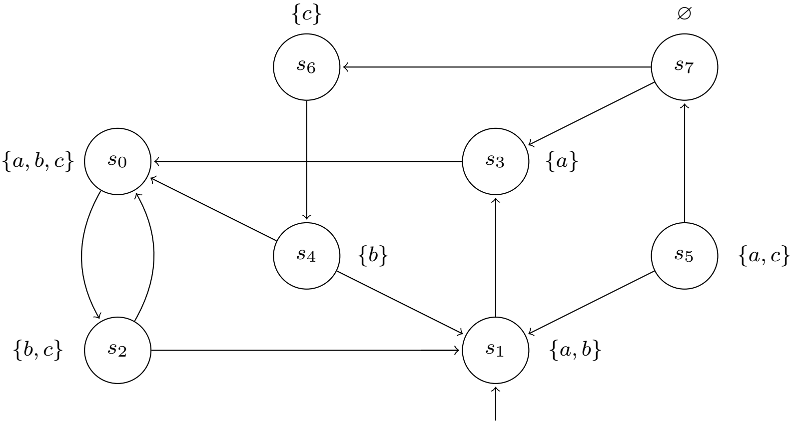
Example

Consider the formula **EX** $(a \leftrightarrow c)$



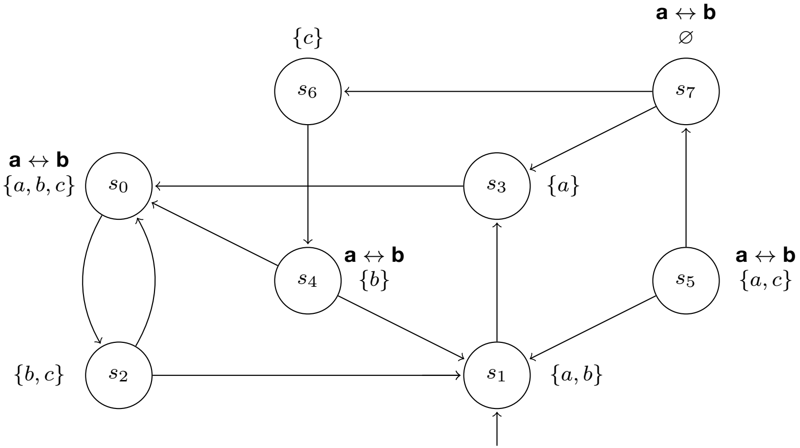
Example: **EX** $(a \leftrightarrow c)$

First label the nodes that satisfy $a \leftrightarrow c$



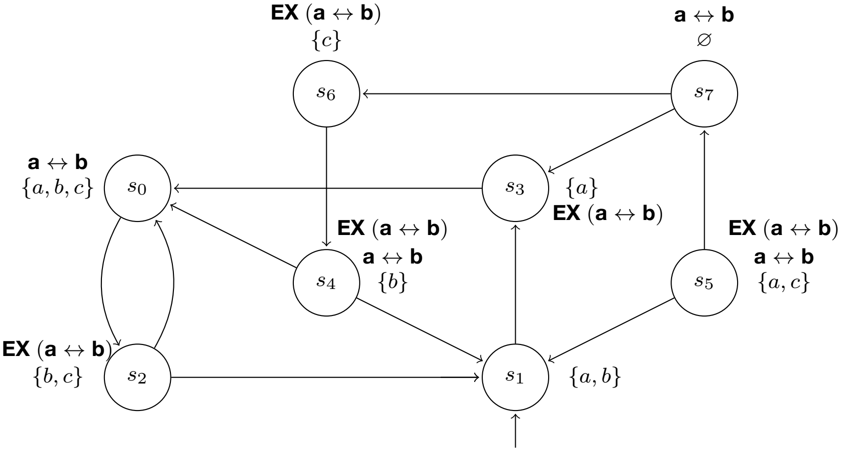
Example: **EX** $(a \leftrightarrow c)$

First label the nodes that satisfy $a \leftrightarrow c$



Example: EX $(a \leftrightarrow c)$

Then the nodes that satisfy **EX** $(a \leftrightarrow c)$



Computing *Sat*($\mathbf{E}$ (ϕ $\mathbf{U}$ ψ))

What to do about $\mathbf{E}$ (ϕ $\mathbf{U}$ ψ)?

Computing $\mathit{Sat}(\mathbf{E}\ (\phi\ \mathbf{U}\ \psi))$

What to do about $\mathbf{E}\ (\phi\ \mathbf{U}\ \psi)$?

We need to find the smallest fixed-point that satisfies:

$$T = \mathit{Sat}(\psi) \cup \{s \in \mathit{Sat}(\phi) \mid \mathbf{Post}(s) \cap T \neq \varnothing\}$$

What to do about **E** (ϕ **U** ψ)?

We need to find the smallest fixed-point that satisfies:

$$T = \textit{Sat}(\psi) \cup \{s \in \textit{Sat}(\phi) \mid \textbf{Post}(s) \cap T \neq \varnothing\}$$

The algorithm will work as follows:

Computing *Sat*(**E** (ϕ **U** ψ))

What to do about **E** (ϕ **U** ψ)?

We need to find the smallest fixed-point that satisfies:

$$T = \textit{Sat}(\psi) \cup \{s \in \textit{Sat}(\phi) \mid \textbf{Post}(s) \cap T \neq \varnothing\}$$

The algorithm will work as follows:

1. First, label the states that satisfy ϕ and ψ

Computing *Sat*(**E** (ϕ **U** ψ))

What to do about **E** (ϕ **U** ψ)?

We need to find the smallest fixed-point that satisfies:

$$T = \textit{Sat}(\psi) \cup \{s \in \textit{Sat}(\phi) \mid \textbf{Post}(s) \cap T \neq \varnothing\}$$

The algorithm will work as follows:

1. First, label the states that satisfy ϕ and ψ
2. Anything labeled ψ is also labeled **E** (ϕ **U** ψ)

Computing $\mathit{Sat}(\mathbf{E}\ (\phi\ \mathbf{U}\ \psi))$

What to do about $\mathbf{E}\ (\phi\ \mathbf{U}\ \psi)$?

We need to find the smallest fixed-point that satisfies:

$$T = \mathit{Sat}(\psi) \cup \{s \in \mathit{Sat}(\phi) \mid \mathbf{Post}(s) \cap T \neq \varnothing\}$$

The algorithm will work as follows:

1. First, label the states that satisfy ϕ and ψ
2. Anything labeled ψ is also labeled $\mathbf{E}\ (\phi\ \mathbf{U}\ \psi)$
3. Then, work backwards using the transition relation

Computing *Sat*(**E** (ϕ **U** ψ))

What to do about **E** (ϕ **U** ψ)?

We need to find the smallest fixed-point that satisfies:

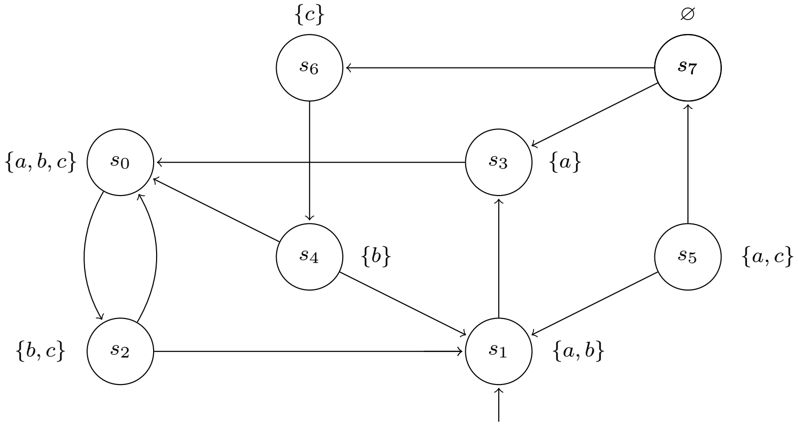
$$T = \textit{Sat}(\psi) \cup \{s \in \textit{Sat}(\phi) \mid \textbf{Post}(s) \cap T \neq \varnothing\}$$

The algorithm will work as follows:

1. First, label the states that satisfy ϕ and ψ
2. Anything labeled ψ is also labeled **E** (ϕ **U** ψ)
3. Then, work backwards using the transition relation
4. Label states that can be reached by a path labeled ϕ

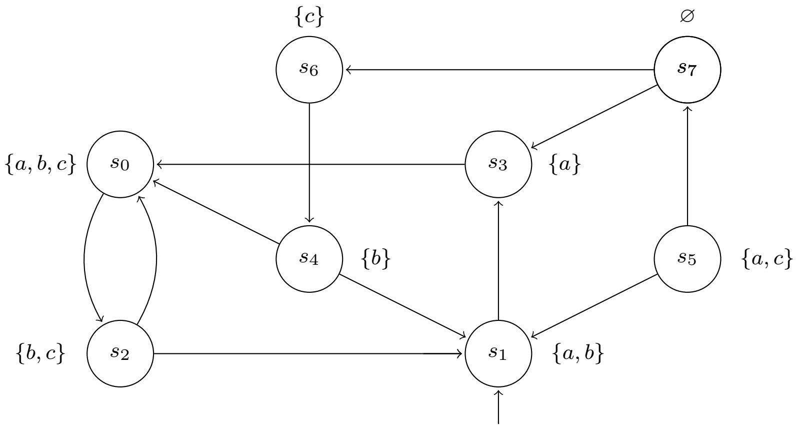
Example

Consider the formula **E** $((b \rightarrow a)$ **U** $c)$



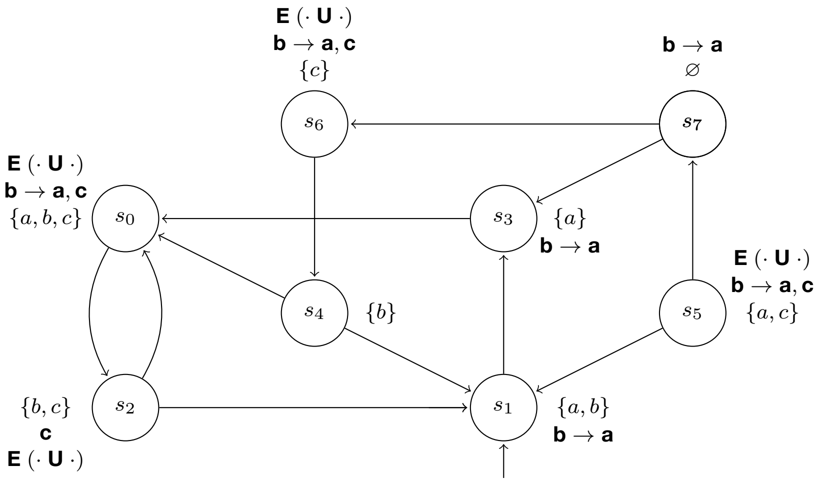
Example

First label states by $b \to a$ and c



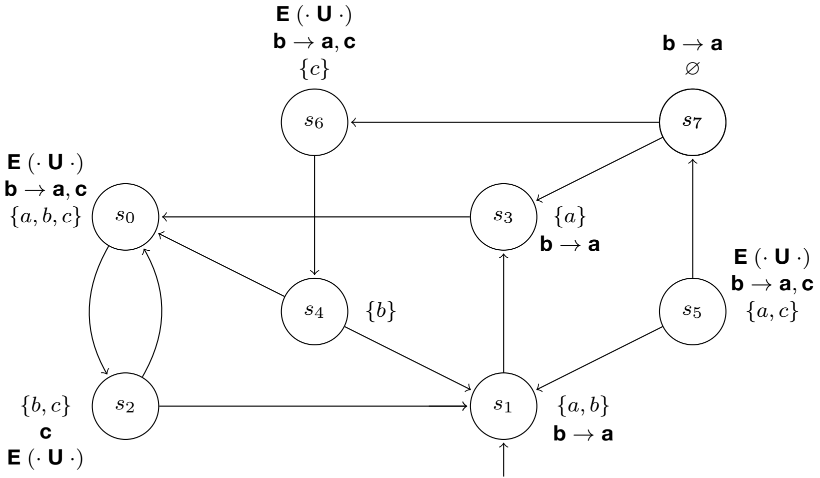
Example

First label states by $b \to a$ and c



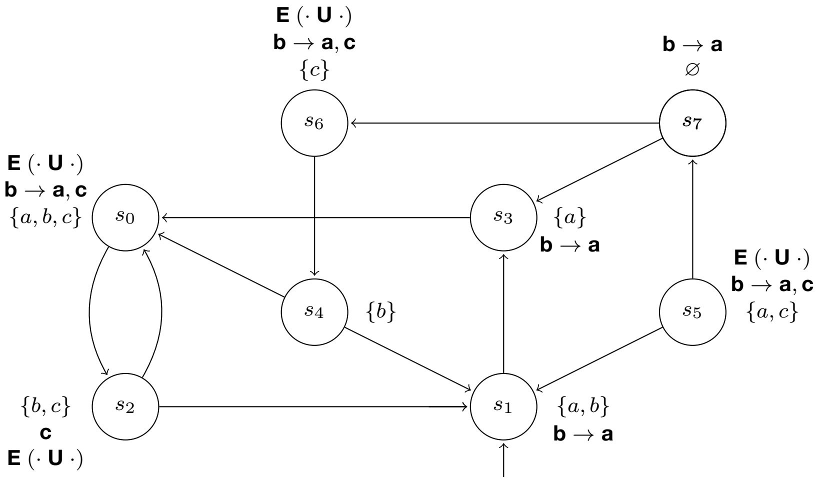
Example

Then work backwards from states labeled **E** $((b \to a)$ **U** $c)$



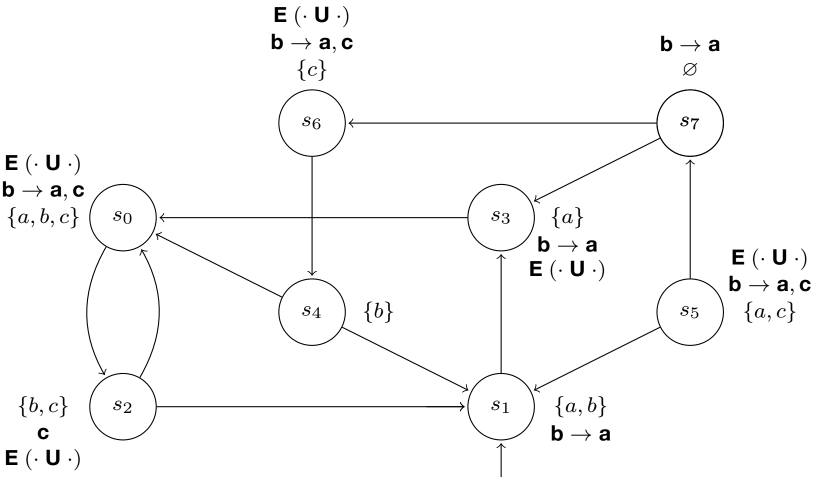
Example

Start with s_0



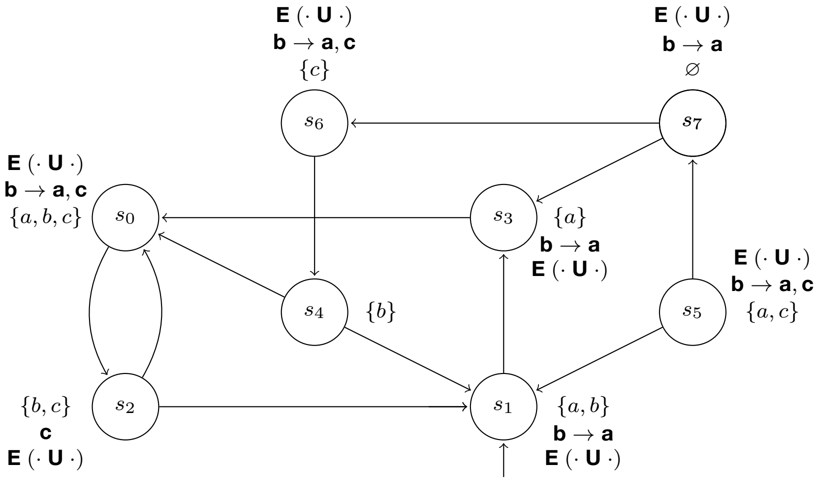
Example

Start with s_0 Then s_3



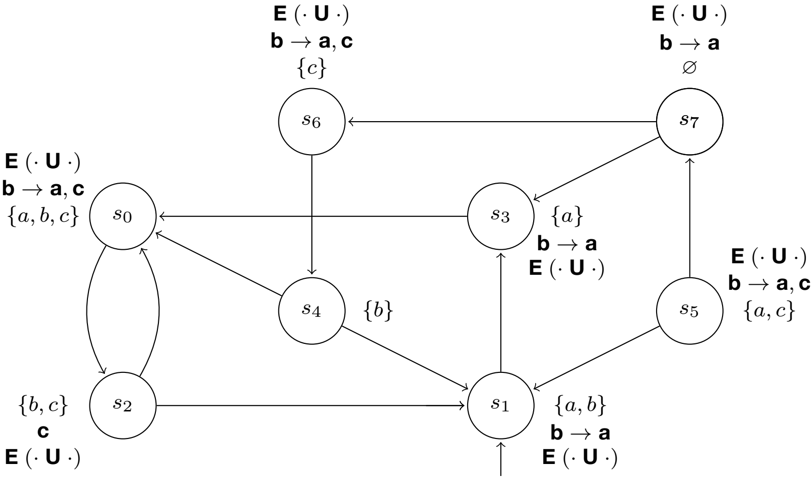
Example

Then s_3



Example

Nothing left to label but s_4; it isn't in *Sat*



Computing $\mathit{Sat}(\mathbf{EG}\ \phi)$

This algorithm is based on **strongly-connected components**

Computing $Sat(\textbf{EG}\ \phi)$

This algorithm is based on **strongly-connected components**

An SCC C is a **maximal** subgraph where:

Computing $Sat(\textbf{EG}\ \phi)$

This algorithm is based on **strongly-connected components**

An SCC C is a **maximal** subgraph where:

- Each node in C is reachable from every other node in C

Computing *Sat*(**EG** ϕ)

This algorithm is based on **strongly-connected components**

An SCC C is a **maximal** subgraph where:

- Each node in C is reachable from every other node in C
- The paths between are contained entirely within C

Computing *Sat*(**EG** ϕ)

This algorithm is based on **strongly-connected components**

An SCC C is a **maximal** subgraph where:

- Each node in C is reachable from every other node in C
- The paths between are contained entirely within C
- C is **nontrivial** if it has more than one node, or a self-loop

Computing *Sat*(**EG** ϕ)

This algorithm is based on **strongly-connected components**

An SCC C is a **maximal** subgraph where:

- Each node in C is reachable from every other node in C
- The paths between are contained entirely within C
- C is **nontrivial** if it has more than one node, or a self-loop

Let M' be obtained from M by deleting all states not satisfying ϕ

Computing *Sat*(**EG** ϕ)

This algorithm is based on **strongly-connected components**

An SCC C is a **maximal** subgraph where:

- Each node in C is reachable from every other node in C
- The paths between are contained entirely within C
- C is **nontrivial** if it has more than one node, or a self-loop

Let M' be obtained from M by deleting all states not satisfying ϕ

Then $M, s \models$ **EG** ϕ iff:

Computing *Sat*(**EG** ϕ)

This algorithm is based on **strongly-connected components**

An SCC C is a **maximal** subgraph where:

- Each node in C is reachable from every other node in C
- The paths between are contained entirely within C
- C is **nontrivial** if it has more than one node, or a self-loop

Let M' be obtained from M by deleting all states not satisfying ϕ

Then $M, s \models$ **EG** ϕ iff:

- $s \in S'$ (i.e., $M, s \models \phi$)

Computing *Sat*(**EG** ϕ)

This algorithm is based on **strongly-connected components**

An SCC C is a **maximal** subgraph where:

- Each node in C is reachable from every other node in C
- The paths between are contained entirely within C
- C is **nontrivial** if it has more than one node, or a self-loop

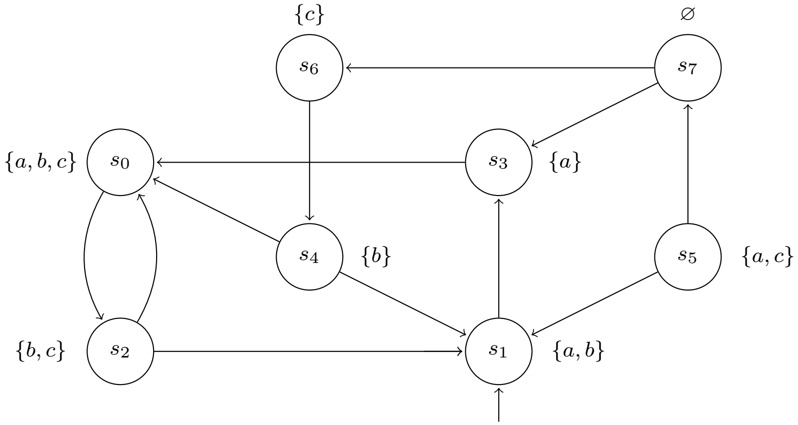
Let M' be obtained from M by deleting all states not satisfying ϕ

Then $M, s \models$ **EG** ϕ iff:

- $s \in S'$ (i.e., $M, s \models \phi$)
- There is a path in M' from s to some t in a nontrivial SCC

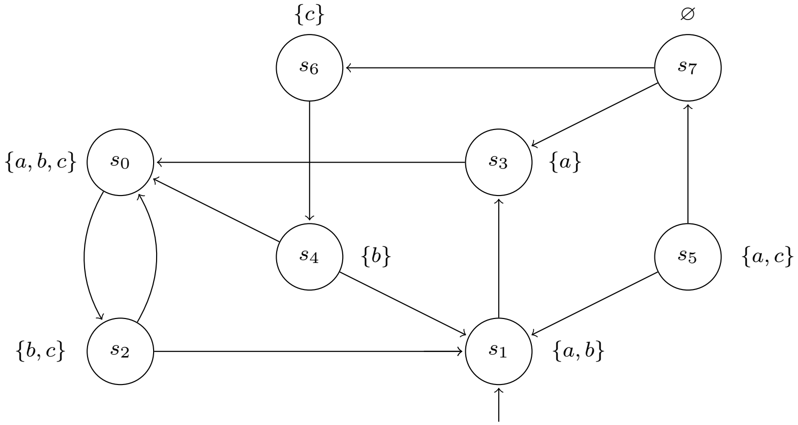
Example

Consider the formula **EG** $(a \vee (b \leftrightarrow c))$



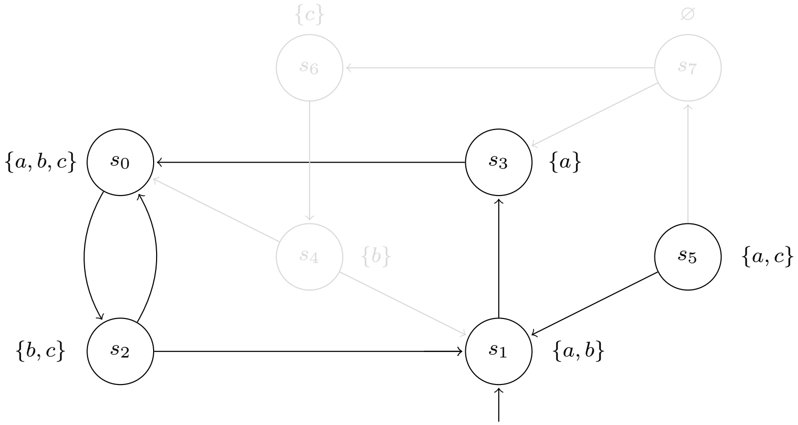
Example

First delete everything not satisfying $a \vee (b \leftrightarrow c)$



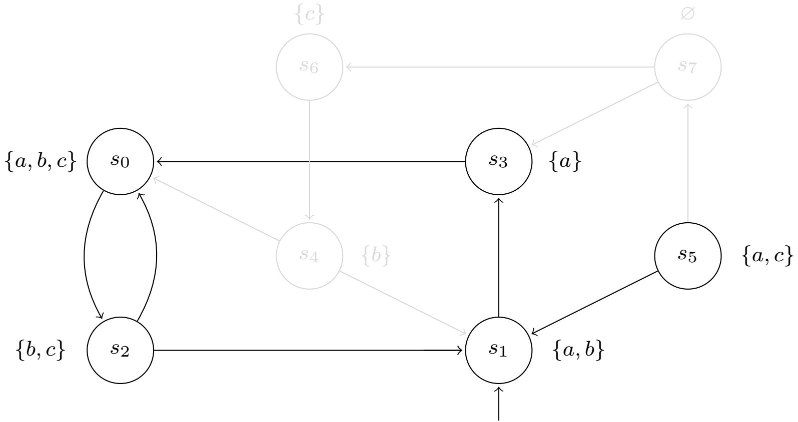
Example

First delete everything not satisfying $a \vee (b \leftrightarrow c)$



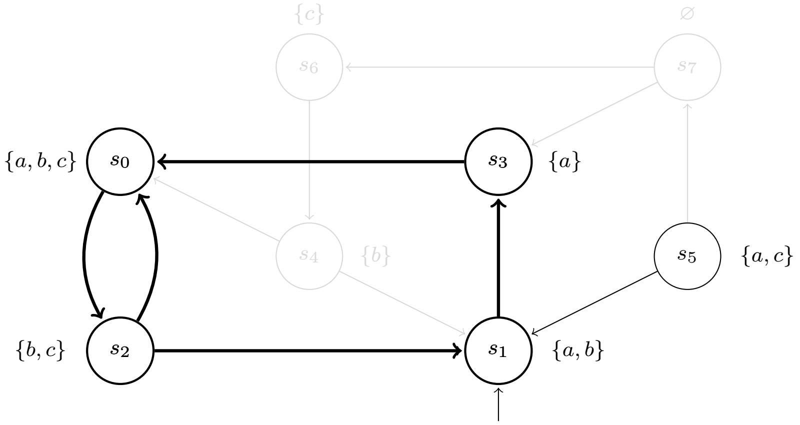
Example

Then find the strongly-connected components



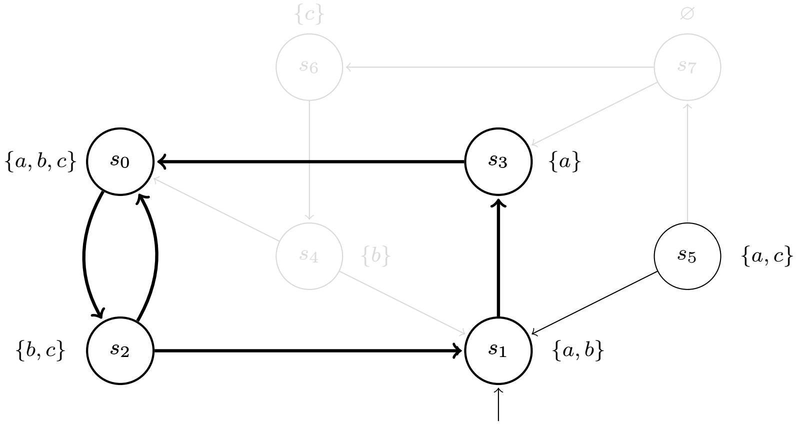
Example

Then find the strongly-connected components



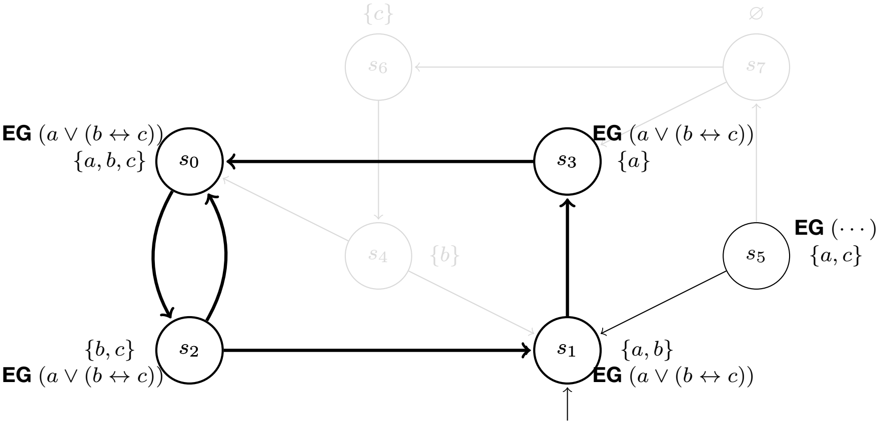
Example

Then label everything that reaches an SCC



Example

Then label everything that reaches an SCC



Complexity of CTL Checking

For the "simple" formulas: *true*$, p, \neg\phi, \phi \wedge \psi,$ **EX** ϕ:

Complexity of CTL Checking

For the "simple" formulas: *true*$, p, \neg\phi, \phi \wedge \psi,$ **EX** ϕ:

- We start with the innermost-nested formulas

Complexity of CTL Checking

For the "simple" formulas: *true*$, p, \neg\phi, \phi \wedge \psi,$ **EX** ϕ:

- We start with the innermost-nested formulas
- Label each state that matches the formula

Complexity of CTL Checking

For the "simple" formulas: *true*$, p, \neg\phi, \phi \wedge \psi,$ **EX** ϕ:

- We start with the innermost-nested formulas
- Label each state that matches the formula
- Each pass takes at most $O(|S| + |R|)$

Complexity of CTL Checking

For the "simple" formulas: *true*$, p, \neg\phi, \phi \wedge \psi,$ **EX** ϕ:

- We start with the innermost-nested formulas
- Label each state that matches the formula
- Each pass takes at most $O(|S| + |R|)$
- We pass once per subformula, so this gives us $O(|\phi| \cdot (|S| + |R|))$

Complexity of CTL Checking

For the "simple" formulas: *true*$, p, \neg\phi, \phi \wedge \psi,$ **EX** ϕ:

- We start with the innermost-nested formulas
- Label each state that matches the formula
- Each pass takes at most $O(|S| + |R|)$
- We pass once per subformula, so this gives us $O(|\phi| \cdot (|S| + |R|))$

E $(\phi$ **U** $\psi)$ is also $O(|S| + |R|)$

Complexity of CTL Checking

For the "simple" formulas: *true*$, p, \neg\phi, \phi \wedge \psi,$ **EX** ϕ:

- We start with the innermost-nested formulas
- Label each state that matches the formula
- Each pass takes at most $O(|S| + |R|)$
- We pass once per subformula, so this gives us $O(|\phi| \cdot (|S| + |R|))$

E $(\phi$ **U** $\psi)$ is also $O(|S| + |R|)$

EG ϕ requires computing strongly-connected components

Complexity of CTL Checking

For the "simple" formulas: *true*$, p, \neg\phi, \phi \wedge \psi,$ **EX** ϕ:

- We start with the innermost-nested formulas
- Label each state that matches the formula
- Each pass takes at most $O(|S| + |R|)$
- We pass once per subformula, so this gives us $O(|\phi| \cdot (|S| + |R|))$

E $(\phi$ **U** $\psi)$ is also $O(|S| + |R|)$

EG ϕ requires computing strongly-connected components

Tarjan's algorithm has complexity $O(|S| + |R|)$

Complexity of CTL Checking

For the "simple" formulas: *true*$, p, \neg\phi, \phi \wedge \psi,$ **EX** ϕ:

- We start with the innermost-nested formulas
- Label each state that matches the formula
- Each pass takes at most $O(|S| + |R|)$
- We pass once per subformula, so this gives us $O(|\phi| \cdot (|S| + |R|))$

E $(\phi$ **U** $\psi)$ is also $O(|S| + |R|)$

EG ϕ requires computing strongly-connected components

Tarjan's algorithm has complexity $O(|S| + |R|)$

Therefore, checking CTL is done in time $O(|\phi| \cdot (|S| + |R|))$

Counterexamples in CTL

When $M, s \not\models \phi$, we want a **counterexample**

Counterexamples in CTL

When $M, s \not\models \phi$, we want a **counterexample**

In general, these are path prefixes that refute ϕ

Counterexamples in CTL

When $M, s \not\models \phi$, we want a **counterexample**

In general, these are path prefixes that refute ϕ

For example, a counterexample of **AF** ϕ is a path π:

Counterexamples in CTL

When $M, s \not\models \phi$, we want a **counterexample**

In general, these are path prefixes that refute ϕ

For example, a counterexample of **AF** ϕ is a path π:

- Beginning with a sequence of $\neg\phi$ states

Counterexamples in CTL

When $M, s \not\models \phi$, we want a **counterexample**

In general, these are path prefixes that refute ϕ

For example, a counterexample of **AF** ϕ is a path π:

- Beginning with a sequence of $\neg\phi$ states
- Ending in a single cycle traversal (with $\neg\phi$)

Counterexamples in CTL

When $M, s \not\models \phi$, we want a **counterexample**

In general, these are path prefixes that refute ϕ

For example, a counterexample of **AF** ϕ is a path π:

- Beginning with a sequence of $\neg\phi$ states
- Ending in a single cycle traversal (with $\neg\phi$)

What is a counterexample for a formula beginning in **E** ?

Counterexamples in CTL

When $M, s \not\models \phi$, we want a **counterexample**

In general, these are path prefixes that refute ϕ

For example, a counterexample of **AF** ϕ is a path π:

- Beginning with a sequence of $\neg\phi$ states
- Ending in a single cycle traversal (with $\neg\phi$)

What is a counterexample for a formula beginning in **E** ?

- In this case, the answer "no" might suffice

Counterexamples in CTL

When $M, s \not\models \phi$, we want a **counterexample**

In general, these are path prefixes that refute ϕ

For example, a counterexample of **AF** ϕ is a path π:

- Beginning with a sequence of $\neg\phi$ states
- Ending in a single cycle traversal (with $\neg\phi$)

What is a counterexample for a formula beginning in **E** ?

- In this case, the answer "no" might suffice
- When **E** ϕ holds, we can provide a **witness**

Counterexamples and Witnesses: **X** operator

This is the simplest case

Counterexamples and Witnesses: **X** operator

This is the simplest case

A counterexample for **AX** ϕ is a pair of states (s, s') where:

Counterexamples and Witnesses: **X** operator

This is the simplest case

A counterexample for **AX** ϕ is a pair of states (s, s') where:

1. $s \in I$

Counterexamples and Witnesses: **X** operator

This is the simplest case

A counterexample for **AX** ϕ is a pair of states (s, s') where:

1. $s \in I$
2. $s' \in$ **Post**(s)

Counterexamples and Witnesses: **X** operator

This is the simplest case

A counterexample for **AX** ϕ is a pair of states (s, s') where:

1. $s \in I$
2. $s' \in$ **Post**(s)
3. $M, s' \not\models \phi$

Counterexamples and Witnesses: **X** operator

This is the simplest case

A counterexample for **AX** ϕ is a pair of states (s, s') where:

1. $s \in I$
2. $s' \in$ **Post**(s)
3. $M, s' \not\models \phi$

A witness for **EX** ϕ is a pair of states (s, s') where:

Counterexamples and Witnesses: **X** operator

This is the simplest case

A counterexample for **AX** ϕ is a pair of states (s, s') where:

1. $s \in I$
2. $s' \in \textbf{Post}(s)$
3. $M, s' \not\models \phi$

A witness for **EX** ϕ is a pair of states (s, s') where:

1. $s \in I$

Counterexamples and Witnesses: **X** operator

This is the simplest case

A counterexample for **AX** ϕ is a pair of states (s, s') where:

1. $s \in I$
2. $s' \in$ **Post**(s)
3. $M, s' \not\models \phi$

A witness for **EX** ϕ is a pair of states (s, s') where:

1. $s \in I$
2. $s' \in$ **Post**(s)

Counterexamples and Witnesses: **X** operator

This is the simplest case

A counterexample for **AX** ϕ is a pair of states (s, s') where:

1. $s \in I$
2. $s' \in$ **Post**(s)
3. $M, s' \not\models \phi$

A witness for **EX** ϕ is a pair of states (s, s') where:

1. $s \in I$
2. $s' \in$ **Post**(s)
3. $M, s' \models \phi$

Counterexamples and Witnesses: **U** operator

A counterexample for **A** $(\phi$ **U** $\psi)$ is a prefix of some π where:

Counterexamples and Witnesses: **U** operator

A counterexample for **A** $(\phi$ **U** $\psi)$ is a prefix of some π where:

1. $M, \pi \models$ **G** $(\phi \wedge \neg\psi)$

Counterexamples and Witnesses: **U** operator

A counterexample for **A** $(\phi$ **U** $\psi)$ is a prefix of some π where:

1. $M, \pi \models$ **G** $(\phi \wedge \neg\psi)$
2. **Or** $M, \pi \models (\phi \wedge \neg\psi)$ **U** $(\neg\phi \wedge \neg\psi$

Counterexamples and Witnesses: **U** operator

A counterexample for **A** $(\phi$ **U** $\psi)$ is a prefix of some π where:

1. $M, \pi \models$ **G** $(\phi \wedge \neg\psi)$
2. **Or** $M, \pi \models (\phi \wedge \neg\psi)$ **U** $(\neg\phi \wedge \neg\psi$

In the first case, the prefix should be of the form:

Counterexamples and Witnesses: **U** operator

A counterexample for **A** $(\phi$ **U** $\psi)$ is a prefix of some π where:

1. $M, \pi \models$ **G** $(\phi \wedge \neg\psi)$
2. **Or** $M, \pi \models (\phi \wedge \neg\psi)$ **U** $(\neg\phi \wedge \neg\psi$

In the first case, the prefix should be of the form:

$$\underbrace{s_1 s_1 \ldots s_{n-1} \underbrace{s_n s'_1 \ldots s'_r}_{\text{cycle}}}_{\text{satisfy } \phi\wedge\neg\psi}$$

Counterexamples and Witnesses: **U** operator

A counterexample for **A** $(\phi$ **U** $\psi)$ is a prefix of some π where:

1. $M, \pi \models$ **G** $(\phi \wedge \neg\psi)$
2. **Or** $M, \pi \models (\phi \wedge \neg\psi)$ **U** $(\neg\phi \wedge \neg\psi$

In the first case, the prefix should be of the form:

$$\underbrace{s_1 s_1 \ldots s_{n-1} \underbrace{s_n s_1' \ldots s_r'}_{\text{cycle}}}_{\text{satisfy } \phi\wedge\neg\psi}$$

In the second, the prefix should be:

Counterexamples and Witnesses: **U** operator

A counterexample for **A** $(\phi$ **U** $\psi)$ is a prefix of some π where:

1. $M, \pi \models$ **G** $(\phi \wedge \neg\psi)$
2. **Or** $M, \pi \models (\phi \wedge \neg\psi)$ **U** $(\neg\phi \wedge \neg\psi$

In the first case, the prefix should be of the form:

$$\underbrace{s_1 s_1 \ldots s_{n-1} \underbrace{s_n s'_1 \ldots s'_r}_{\text{cycle}}}_{\text{satisfy } \phi\wedge\neg\psi}$$

In the second, the prefix should be:

$$\underbrace{s_1 s_1 \ldots s_{n-1}}_{\text{satisfy } \phi\wedge\neg\psi} s_n \quad \text{with } M, s_n \models \neg\phi \wedge \neg\psi$$

Counterexamples and Witnesses: **U** operator

A witness for **A** $(\phi$ **U** $\psi)$ is a prefix of some π where:

Counterexamples and Witnesses: **U** operator

A witness for **A** $(\phi$ **U** $\psi)$ is a prefix of some π where:

1. $\pi = s_0 s_1 \ldots s_n$

Counterexamples and Witnesses: **U** operator

A witness for **A** $(\phi$ **U** $\psi)$ is a prefix of some π where:

1. $\pi = s_0 s_1 \ldots s_n$
2. $M, s_i \models \phi$ for all $0 \leq i < n$

Counterexamples and Witnesses: **U** operator

A witness for **A** $(\phi$ **U** $\psi)$ is a prefix of some π where:

1. $\pi = s_0 s_1 \ldots s_n$
2. $M, s_i \models \phi$ for all $0 \leq i < n$
3. $M, s_n \models \psi$

Counterexamples and Witnesses: **U** operator

A witness for **A** $(\phi$ **U** $\psi)$ is a prefix of some π where:

1. $\pi = s_0 s_1 \dots s_n$
2. $M, s_i \models \phi$ for all $0 \leq i < n$
3. $M, s_n \models \psi$

Witnesses are generated by backward search from ψ states

Counterexamples and Witnesses: U operator

A witness for **A** $(\phi$ **U** $\psi)$ is a prefix of some π where:

1. $\pi = s_0 s_1 \ldots s_n$
2. $M, s_i \models \phi$ for all $0 \leq i < n$
3. $M, s_n \models \psi$

Witnesses are generated by backward search from ψ states

Counterexamples are generated by backwards search from:

Counterexamples and Witnesses: **U** operator

A witness for **A** $(\phi$ **U** $\psi)$ is a prefix of some π where:

1. $\pi = s_0 s_1 \ldots s_n$
2. $M, s_i \models \phi$ for all $0 \leq i < n$
3. $M, s_n \models \psi$

Witnesses are generated by backward search from ψ states

Counterexamples are generated by backwards search from:

1. SCCs satisfying $\phi \wedge \neg\psi$

Counterexamples and Witnesses: **U** operator

A witness for **A** $(\phi$ **U** $\psi)$ is a prefix of some π where:

1. $\pi = s_0 s_1 \ldots s_n$
2. $M, s_i \models \phi$ for all $0 \leq i < n$
3. $M, s_n \models \psi$

Witnesses are generated by backward search from ψ states

Counterexamples are generated by backwards search from:

1. SCCs satisfying $\phi \wedge \neg\psi$
2. **Or** $\neg\phi \wedge \neg\psi$

Counterexamples and Witnesses: **G** operator

A counterexample for **AG** ϕ is a prefix of some π where:

Counterexamples and Witnesses: **G** operator

A counterexample for **AG** ϕ is a prefix of some π where:

1. $\pi = s_0 s_1 \dots s_n$

Counterexamples and Witnesses: **G** operator

A counterexample for **AG** ϕ is a prefix of some π where:

1. $\pi = s_0 s_1 \ldots s_n$
2. $M, s_i \models \phi$ for all $0 \leq i < n$

Counterexamples and Witnesses: **G** operator

A counterexample for **AG** ϕ is a prefix of some π where:

1. π = $s_0 s_1 \ldots s_n$
2. $M, s_i \models \phi$ for all $0 \leq i < n$
3. $M, s_n \models \neg\phi$

Counterexamples and Witnesses: **G** operator

A counterexample for **AG** ϕ is a prefix of some π where:

1. $\pi = s_0 s_1 \ldots s_n$
2. $M, s_i \models \phi$ for all $0 \leq i < n$
3. $M, s_n \models \neg\phi$

A witness for **EG** ϕ is a prefix of the form:

$$\underbrace{s_0 s_1 \ldots s_n s'_1 \ldots s'_r}_{\text{satisfies } \phi} \quad \text{where } s_n = s'_r$$

Counterexamples and Witnesses: **G** operator

A counterexample for **AG** ϕ is a prefix of some π where:

1. $\pi = s_0 s_1 \ldots s_n$
2. $M, s_i \models \phi$ for all $0 \leq i < n$
3. $M, s_n \models \neg\phi$

A witness for **EG** ϕ is a prefix of the form:

$$\underbrace{s_0 s_1 \ldots s_n s'_1 \ldots s'_r}_{\text{satisfies } \phi} \quad \text{where } s_n = s'_r$$

Counterexamples are found by backward search

Counterexamples and Witnesses: **G** operator

A counterexample for **AG** ϕ is a prefix of some π where:

1. $\pi = s_0 s_1 \ldots s_n$
2. $M, s_i \models \phi$ for all $0 \leq i < n$
3. $M, s_n \models \neg\phi$

A witness for **EG** ϕ is a prefix of the form:

$$\underbrace{s_0 s_1 \ldots s_n s'_1 \ldots s'_r}_{\text{satisfies } \phi} \quad \text{where } s_n = s'_r$$

Counterexamples are found by backward search

Witnesses are found by looking for cycles that satisfy ϕ, backward search

Next Lecture

- Continue discussing model checking
- Infinite automata
- LTL model checking
- Dealing with state explosion