Software Verification

Lecture 11: Model Checking

Carlo A. Furia

Program Verification: the very idea

P: a program

S: a specification

```
max (a, b: INTEGER): INTEGER is
        do
                if a > b then
                        Result := a
                else
                        Result := b
                end
        end
```

```
require
        true

ensure
        Result >= a
        Result >= b
```

Does $P \models S$ hold?

The Program Verification problem:

- Given: a program P and a specification S
- Determine: if every execution of P, for every value of input parameters, satisfies S

Why is Verification Difficult?

The very nature of universal (Turing-complete) computation entails the impossibility of deciding automatically the program verification problem.

$$\begin{tikzcd}
\text{P: a program} \arrow[d, Leftrightarrow] & \text{S: a specification} \arrow[d, Leftrightarrow] \\
\text{TM(P): a Turing machine} & \text{F(S): a first-order formula}
\end{tikzcd}$$

Does $TM(P) \models F(S)$ hold?

UNDECIDABLE

Decidability vs. Expressiveness Trade-Off

If we restrict the expressiveness of:

- the computational model

and/or

- the specification language

the verification problem may become decidable

Does $P \models S$ hold?

Def. Expressiveness: capability of describing extensive classes of:

- computations
- properties

Verification of Finite-state Programs

Verification of Finite-state Programs

In Model Checking we typically assume:

- finite-state programs
  - every variable has finite domain
  - bounded dynamic allocation
  - bounded recursion
- monadic first-order logic
  - restricted first-order logic fragment where the ordering of state values during a computation can be expressed

P: a finite-state program   S: a monadic first-order specification

Does $P \vDash S$ hold?

DECIDABLE

Verification of Finite-state Programs

In Model Checking we typically assume:

- finite-state programs

  equivalently: finite-state automata of some kind
- monadic first-order logic

  equivalently: temporal logic of some kind

P: a program $\Updownarrow$ FSA(P): a finite-state automaton

S: a specification $\Updownarrow$ TL(S): a temporal logic formula

Does $P \vDash S$ hold?

DECIDABLE

Model-checking in Pictures

```
is_locked: BOOLEAN

toggle_lock:

    do

      is_locked := not is_locked

    end
```

P: a program

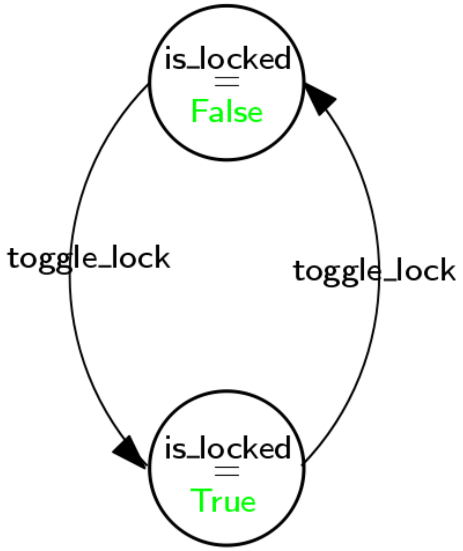
⇕

FSA(P): a finite-state automaton

S: a specification

⇕

TL(S): a temporal logic formula



|= [] (toggle_lock ⇒ X toggle_lock)

Finite-state Programs in the Real World

Can finite-state models capture significant aspects of real programs? Yes!

A few examples:

- Behavior of hardware
  - inherently finite-state
- Concurrency aspects
  - access to critical regions, scheduling of processes, ...
- Security aspects
  - access policies, protocols, ...
- Reactive systems
  - ongoing interaction between software and physical environment

Is the Abstraction Correct?

How to guarantee that the finite-state abstraction of an infinite-state program is accurate?

- In hardware verification, the real system is finite-state, so no abstraction is needed
- The finite-state model can be built and verified before the real implementation is produced
  - A formal high-level model: increased confidence in some key features of the system under development
  - Model-driven development: the implementation is derived (almost) automatically from the high-level finite-state model

Is the Abstraction Correct?

How to guarantee that the finite-state abstraction of an infinite-state program is accurate?

- Software model-checking: the abstraction is built automatically and refined iteratively until we can guarantee that it is an accurate model of the real implementation for the properties under verification

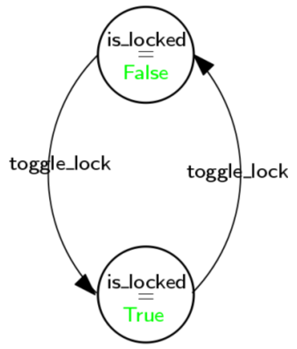
The Model-Checking Paradigm

The Model-Checking Paradigm

The Model Checking problem:

- Given: a finite-state automaton A and
  a temporal-logic formula F
- Determine: if every run of A satisfies F or not
  - if not, provide a counterexample:
    a run of A where F does not hold

A: a finite-state automaton

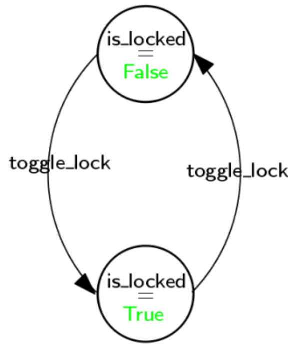


F: a temporal-logic formula

$\models$ [] (toggle_lock $\Rightarrow$ X toggle_lock)

The Model-Checking Paradigm

A: a finite-state automaton



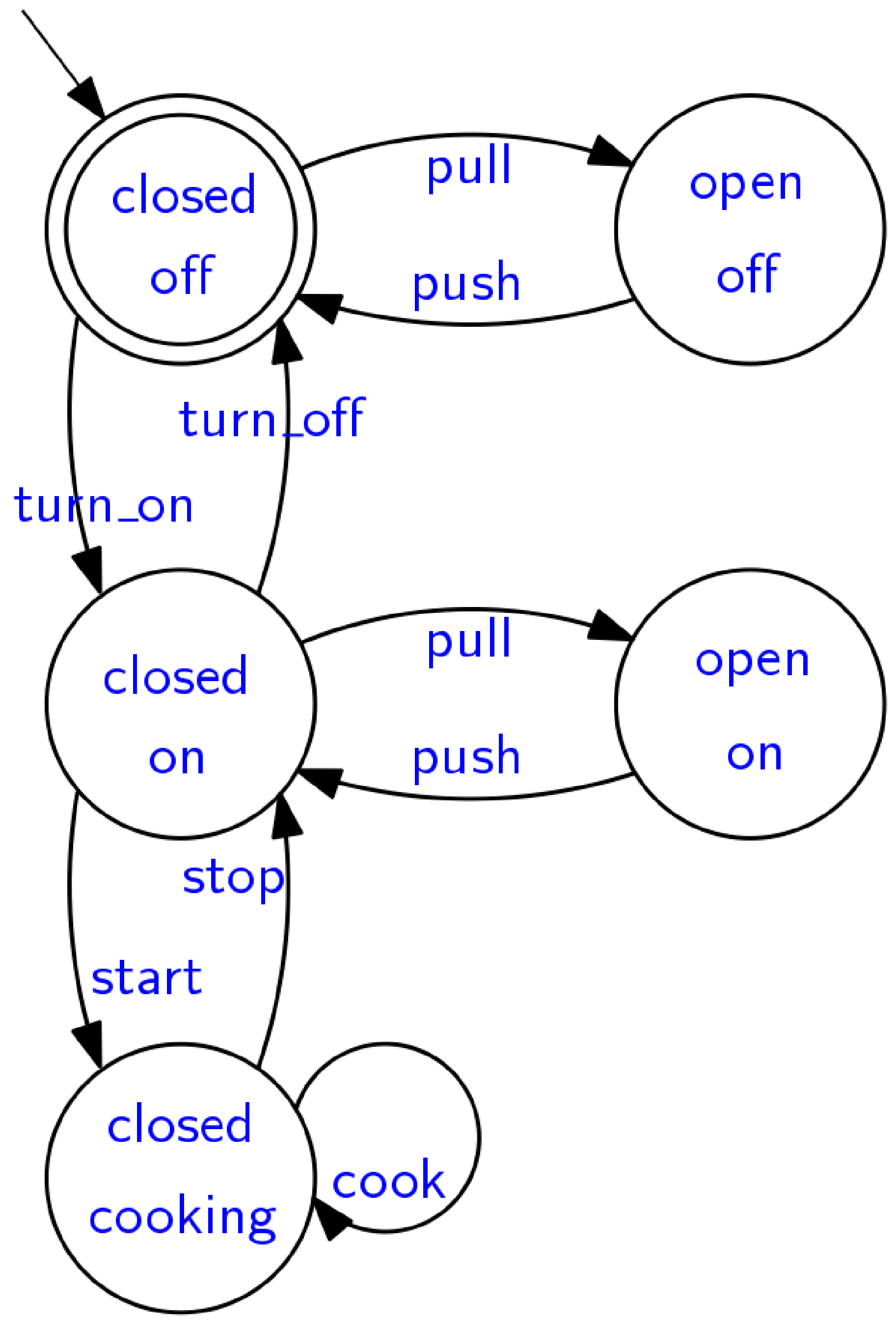
F: a temporal-logic formula

⊨ [] (toggle_lock ⇔ X toggle_lock)

Different choices are possible for the kinds of automaton and of formula.

- We now describe more details for linear-time model-checking where:
  - A is a (nondeterministic) finite-state automaton
  - F is a propositional linear temporal logic formula

Finite State Automata: Syntax



Finite State Automata: Syntax

Def. Nondeterministic Finite State Automaton (FSA):

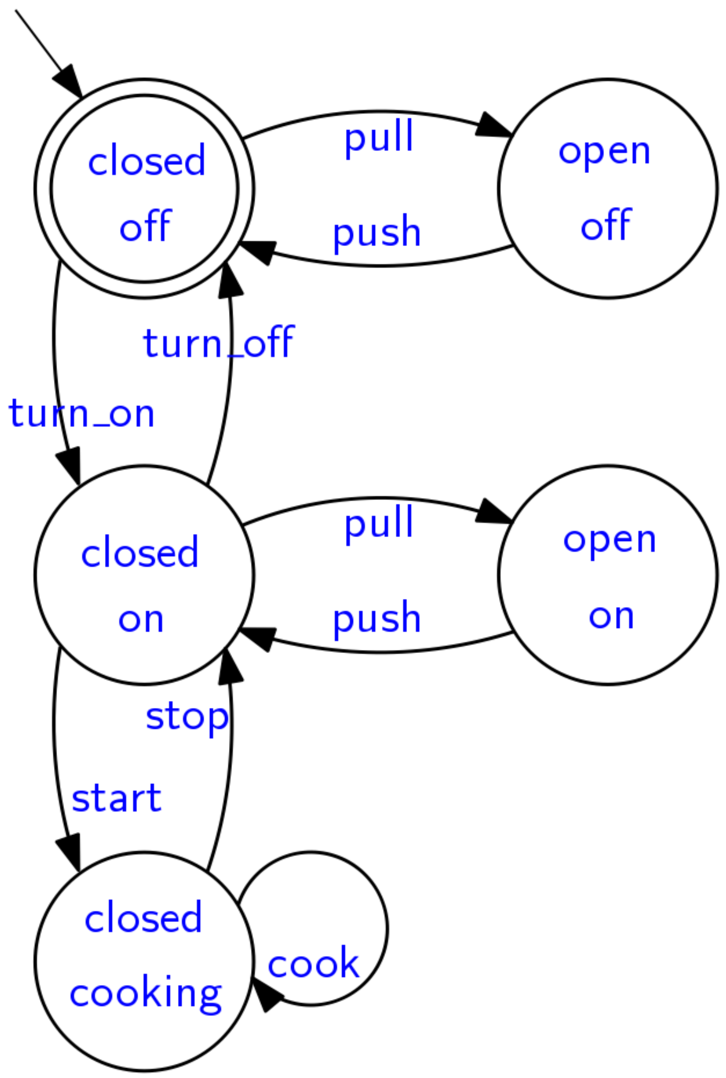
a tuple $[\Sigma, S, I, \rho, F]$:

- Σ: finite nonempty (input) alphabet
- S: finite nonempty set of states
- $I \subseteq S$: set of initial states
- $F \subseteq S$: set of accepting states
- ρ: $S \times \Sigma \rightarrow 2^S$: transition function

Finite State Automata: Syntax

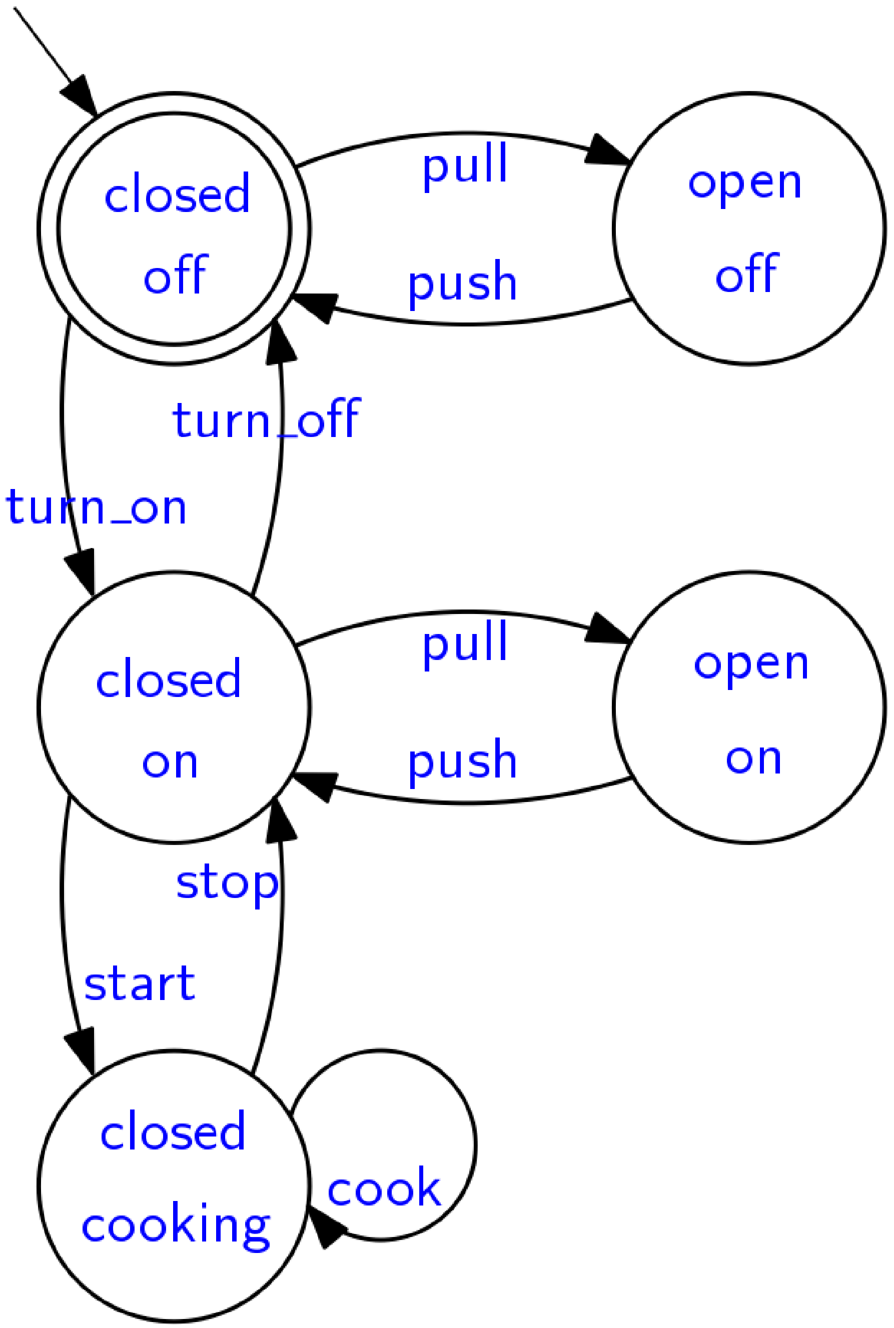
Def. Nondeterministic Finite State Automaton (FSA):
a tuple [Σ, S, I, ρ, F]:

- Σ: finite nonempty (input) alphabet
- S: finite nonempty set of states
- I ⊆ S: set of initial states
- F ⊆ S: set of accepting states
- ρ: S x Σ → 2^S: transition function



- Σ = { pull, push, turn_on, turn_off, start, stop, cook }
- S = { closed-off, open-off, closed-on, open-on, closed-cooking }
- I = { closed-off }
- F = { closed-off }
- ρ(closed-off, turn_on) = { closed-on }
- ρ(..., ...) = ...
  - Deterministic, in this example ("microwave oven")

Finite State Automata: Semantics



Accepting run

r = closed-off closed-on closed-cooking closed-cooking closed-on closed-off

over input word

w = turn_on start cook stop turn_off

Rejecting run

r' =closed-off open-off closed-off closed-on

over input word

w' = pull push turn_on

Finite State Automata: Semantics

Def. An accepting run of an FSA $A=[\Sigma, S, I, \rho, F]$ over input word $w = w(1)\ w(2) \ldots w(n) \in \Sigma^*$ is a sequence $r = r(0)\ r(1)\ r(2) \ldots r(n) \in S^*$ of states such that:

- it starts from an initial state: $r(0) \in I$
- it ends in an accepting state: $r(n) \in F$
- it respects the transition function:

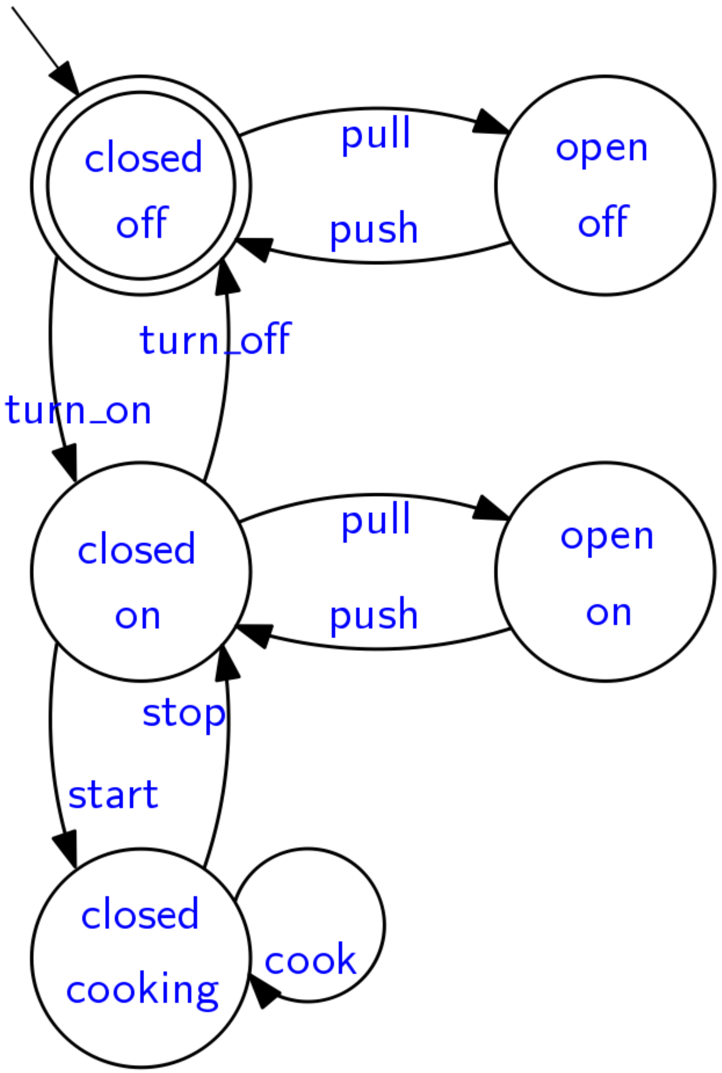
  $r(i+1) \in \rho(r(i), w(i))$ for all $0 \leq i < n$

Finite State Automata: Semantics

Def. An accepting run of an FSA A=[Σ, S, I, ρ, F]
over input word w = w(1) w(2) … w(n) ∈ Σ*
is a sequence r = r(0) r(1) r(2) … r(n)∈ S*
of states such that:

–	it starts from an initial state:	r(0) ∈ I
–	it ends in an accepting state:	r(n) ∈ F
–	it respects the transition function: r(i+1) ∈ ρ(r(i), w(i))	for all 0 ≤ i < n

- Accepting run
  r = closed-off closed-on closed-cooking
  closed-cooking closed-on closed-off
- Over input word
  w = turn_on start cook stop turn_off
- In practice: any path on the directed graph that starts in an initial state and ends in an accepting state



Finite State Automata: Semantics

Def. Any FSA $A=[\Sigma, S, I, \rho, F]$ defines

a set of input words $\langle A \rangle$:

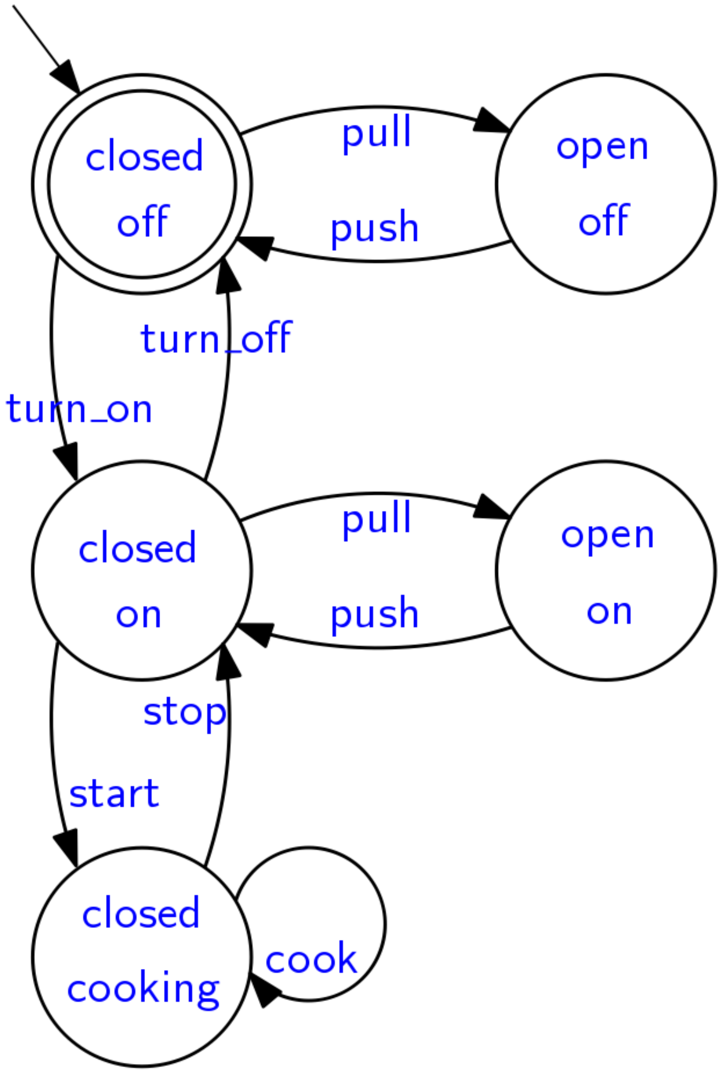
$\langle A \rangle \triangleq \{ w \in \Sigma^* \mid$ there is an accepting run of A over w $\}$

$\langle A \rangle$ is called the language of A

Finite State Automata: Semantics

Def. Any FSA $A=[\Sigma, S, I, \rho, F]$ defines
a set of input words $\langle A \rangle$:
$\langle A \rangle \triangleq \{ w \in \Sigma^* \mid$ there is an accepting run of A over $w \}$

$\langle A \rangle$ is called the language of A



With regular expressions:

$\langle A \rangle$ = ((pull push)* (turn_on
(pull push)*
(start cook* stop)*
(pull push)*
turn_off)*)*

Linear Temporal Logic: Syntax

Def. Propositional Linear Temporal Logic (LTL) formulae are defined by the grammar:

$$F ::= p \mid \neg F \mid F \wedge G \mid X\, F \mid F\, U\, G$$

with $p \in P$ any atomic proposition from a fixed set P.

Temporal (modal) operators:

- next: $X\, F$
- until: $F\, U\, G$
- release: $F\, R\, G \triangleq \neg(\neg F\, U\, \neg G)$
- eventually: $\langle\rangle F \triangleq True\, U\, F$
- always: $[]\, F \triangleq \neg \langle\rangle \neg F$

Propositional connectives:

- not: $\neg F$
- and: $F \wedge G$
- or: $F \vee G \triangleq \neg(\neg F \wedge \neg G)$
- implies: $F \Rightarrow G \triangleq \neg F \vee G$
- iff: $F \Leftrightarrow G \triangleq (F \Rightarrow G) \wedge (G \Rightarrow F)$

Linear Temporal Logic: Syntax

Def. Propositional Linear Temporal Logic (LTL) formulae are defined by the grammar:

$$F ::= p \mid \neg F \mid F \wedge G \mid X\, F \mid F\, U\, G$$

with $p \in P$ any atomic proposition from a fixed set P.

$$[]\,(\,start \Rightarrow X\,(cook\ U\ stop)\,)$$

Linear Temporal Logic: Semantics

- [] (start)
- X (cook)
- [] (X cook)
- X ([] cook)
- stop ∧ start

Linear Temporal Logic: Semantics

- [] (start)

  start, start, start, ...
- X (cook)

  [any], cook, [any], ...
- [] (X cook)

  ε (the empty word)
- X ([] cook)

  [any], cook, cook, cook, ...
- stop ∧ start

  ∅

  (assuming Σ defined before as alphabet)

Linear Temporal Logic: Semantics

Def. A word $w = w(1)\ w(2) \dots w(n) \in P^*$
satisfies an LTL formula F
at position $1 \leq i \leq n$, denoted $w, i \vDash F$,
under the following conditions:

- $w, i \vDash p$ iff $p = w(i)$
- $w, i \vDash \neg F$ iff $w, i \vDash F$ does not hold
- $w, i \vDash F \land G$ iff both $w, i \vDash F$ and $w, i \vDash G$ hold
- $w, i \vDash X\ F$ iff $i < n$ and $w, i+1 \vDash F$
  - i.e., F holds in the next step
- $w, i \vDash F\ U\ G$ iff for some $i \leq j \leq n$ it is: $w, j \vDash G$ and for all $i \leq k < j$ it is $w, k \vDash F$
  - i.e., F holds until G will hold

Linear Temporal Logic: Semantics

For derived operators:

- w, i ⊨ <> F iff for some i ≤ j ≤ n it is: w, j ⊨ F
  - i.e., F holds eventually (in the future)
- w, i ⊨ [] F iff for all i ≤ j ≤ n it is: w, j ⊨ F
  - i.e., F holds always (in the future)

Linear Temporal Logic: Semantics

Def. Satisfaction:

$$w \models F \quad \triangleq \quad w, 1 \models F$$

i.e., word w satisfies formula F initially

Def. Any LTL formula F defines a set of words $\langle F \rangle$:

$$\langle F \rangle \triangleq \{\, w \in P^* \mid \quad w \models F \,\}$$

$\langle F \rangle$ is called the language of F

Linear Temporal Logic: Semantics

Def. Any LTL formula F defines a set of words ⟨F⟩:

$$\langle F \rangle \triangleq \{ w \in P^* \mid w \models F \}$$

⟨F⟩ is called the language of F

$\langle \Box \text{ start} \rangle = \text{start}, \text{start}, \text{start}, \ldots$

Verification as Emptiness Checking

The Model Checking problem:

- Given: a finite-state automaton A and a temporal-logic formula F
- Determine: if every run of A satisfies F or not
  - if not, also provide a counterexample: a run of A where F does not hold

$$\begin{tikzcd}
A\text{: a finite-state automaton} \arrow[d, Leftrightarrow] & \overset{?}{\models} & F\text{: a temporal-logic formula} \arrow[d, Leftrightarrow] \\
\langle A \rangle = \text{words accepted by } A & & \langle F \rangle = \text{words satisfying } F
\end{tikzcd}$$

Verification as Emptiness Checking

A: a finite-state automaton $\overset{?}{\models}$ F: a temporal-logic formula

$\Updownarrow$ $\Updownarrow$

⟨A⟩ = words accepted by A ⟨F⟩ = words satisfying F

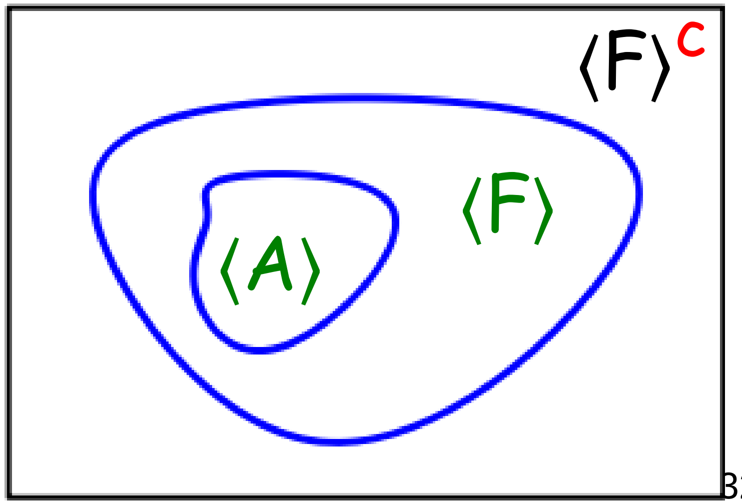
A ⊨ F means: " every accepting run of A produces a word that satisfies F "

A ⊨ F iff: $w \in \langle A\rangle$ implies $w \in \langle F\rangle$

iff: $\langle A\rangle \subseteq \langle F\rangle$

iff: $\langle A\rangle \cap \langle F\rangle^c = \emptyset$

iff: $\langle A\rangle \cap \langle \neg F\rangle = \emptyset$



Automata-theoretic Model Checking

A semantic view of the Model Checking problem:

- Given: a finite-state automaton A
  and a temporal-logic formula F
- if $\langle A \rangle \cap \langle \neg F \rangle$ is empty then every run of A satisfies F
- if $\langle A \rangle \cap \langle \neg F \rangle$ is not empty then some run of A does not satisfy F
  - any member of the nonempty intersection $\langle A \rangle \cap \langle \neg F \rangle$ is a counterexample

Automata-theoretic Model Checking

How to check $\langle A \rangle \cap \langle \neg F \rangle = \emptyset$ algorithmically (given A, F)?

Combination of three different algorithms:

- LTL2FSA: given LTL formula F build automaton $a(F)$ such that $\langle F \rangle = \langle a(F) \rangle$
- FSA-Intersection: given automata A, B build automaton C such that $\langle A \rangle \cap \langle B \rangle = \langle C \rangle$
- FSA-Emptiness: given automaton A check whether $\langle A \rangle = \emptyset$ is the case

LTL2FSA: from LTL to FSA

Given an LTL formula F, it is always possible to build automatically an FSA a(F) that accepts precisely the same words that satisfy F.

There are various algorithms to achieve this, with various degrees of sophistication and efficiency. Let us skip the details and just demonstrate the idea on an example.

LTL2FSA: from LTL to FSA

$$\Box\ (\ start \Rightarrow X\ (cook\ U\ stop)\ )$$

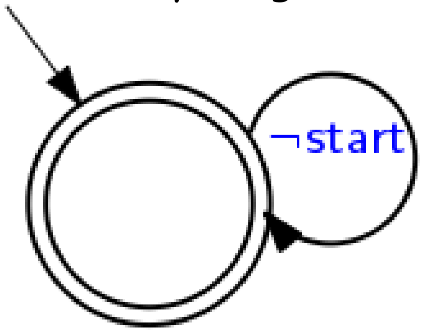
- Always:
  - when start occurs:
    - stop will occur in the future and
    - cook holds until the occurrence of stop

LTL2FSA: from LTL to FSA

$[]\ (\text{start} \Rightarrow X\ (\text{cook}\ U\ \text{stop})\ )$

- Always:
  - when start occurs:
    - stop will occur in the future and
    - cook holds until the occurrence of stop

As long as start does not occur, everything's fine.

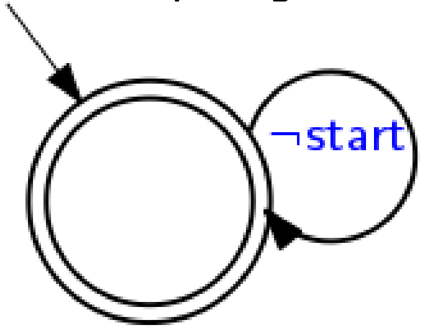


LTL2FSA: from LTL to FSA

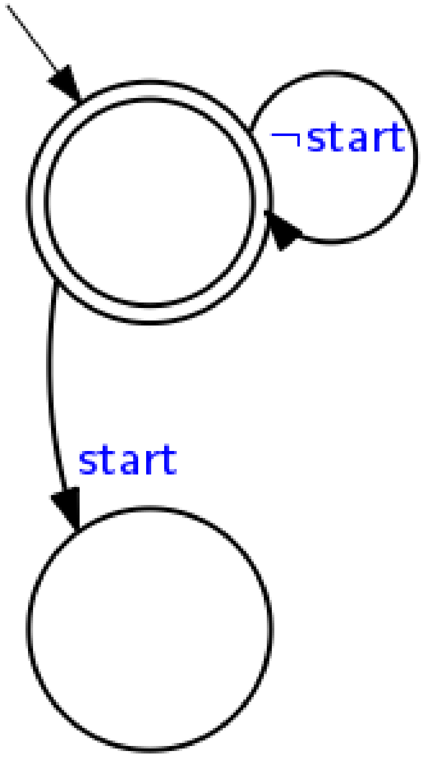
$$[]\ (\ start \Rightarrow X\ (cook\ U\ stop)\ )$$

- Always:
  - when start occurs:
    - stop will occur in the future and
    - cook holds until the occurrence of stop

As long as start does not occur, everything's fine.



start occurs: move to a different (non-accepting) state and start monitoring.

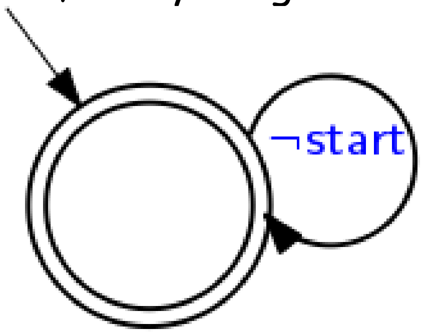


LTL2FSA: from LTL to FSA

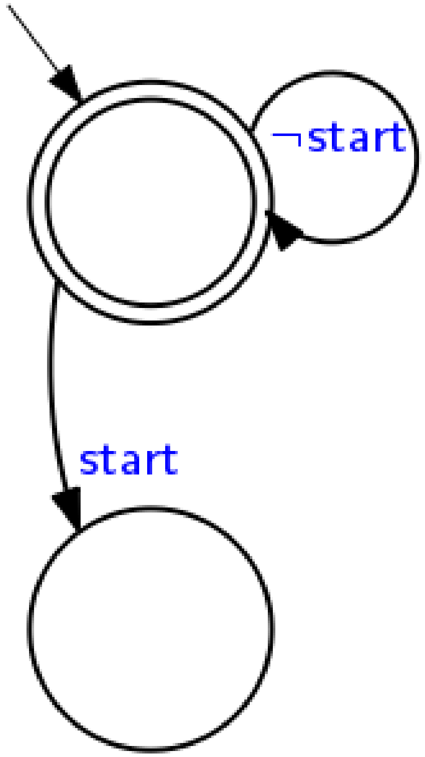
$$[]\,(\, start \Rightarrow X\, (cook\ U\ stop)\,)$$

- Always:
  - when start occurs:
    - stop will occur in the future and
    - cook holds until the occurrence of stop

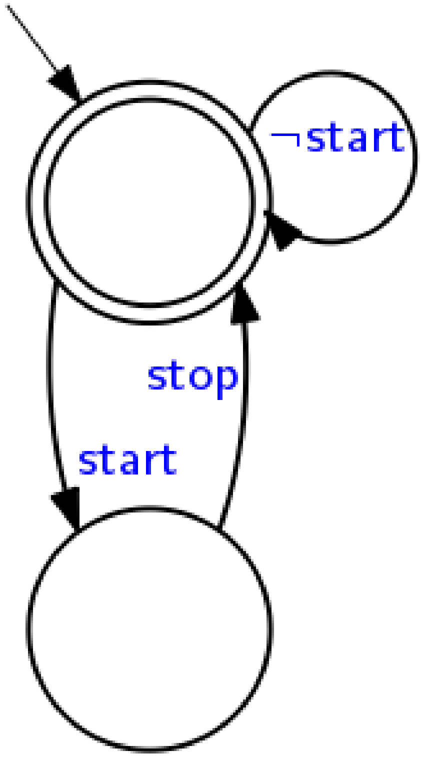
As long as start does not occur, everything's fine.



start occurs: move to a different (non-accepting) state and start monitoring.



stop must occur in the future for things to be fine.

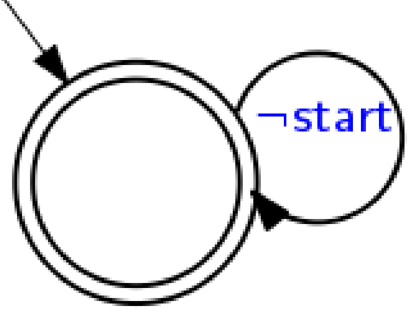


LTL2FSA: from LTL to FSA

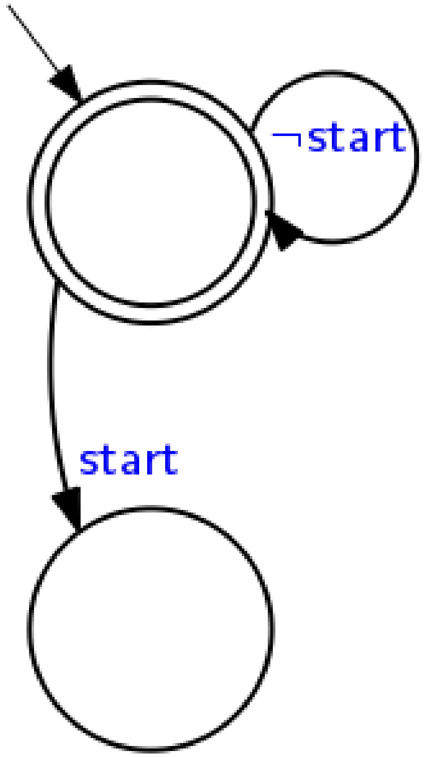
[] (start ⇒ X (cook U stop))

- Always:
  - when start occurs:
    - stop will occur in the future and
    - cook holds until the occurrence of stop

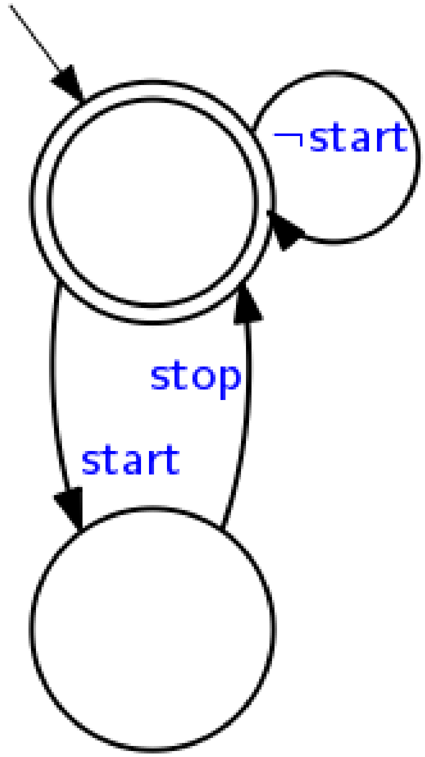
As long as start does not occur, everything's fine.



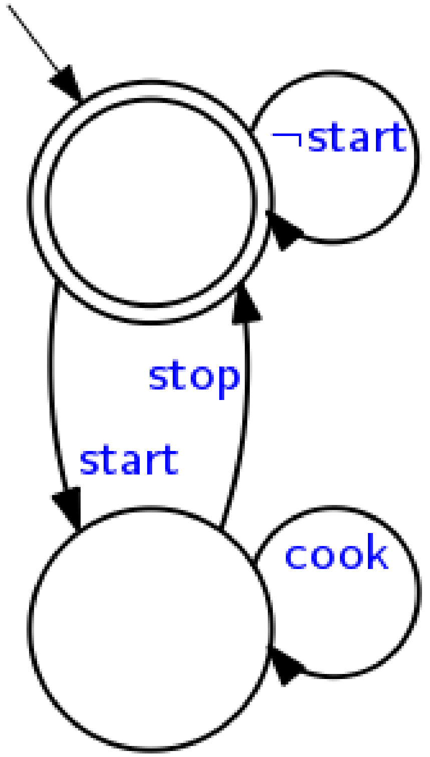
start occurs: move to a different (non-accepting) state and start monitoring.



stop must occur in the future for things to be fine.



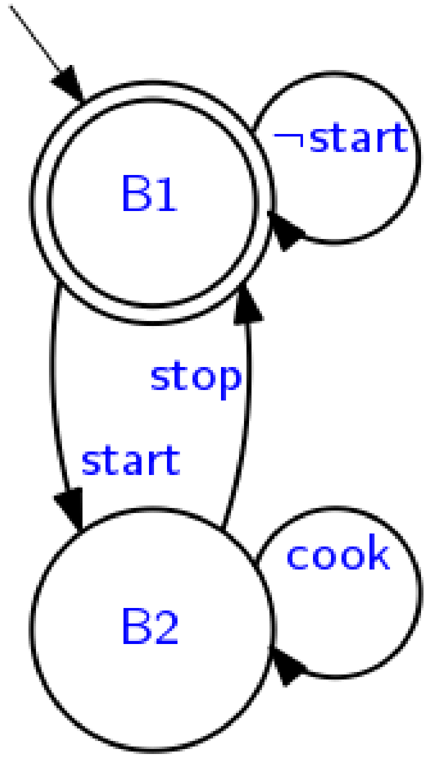
cook can occur before stop does.



LTL2FSA: from LTL to FSA

[] (start ⇒ X (cook U stop))

- Always:
  - when start occurs:
    - stop will occur in the future and
    - cook holds until the occurrence of stop



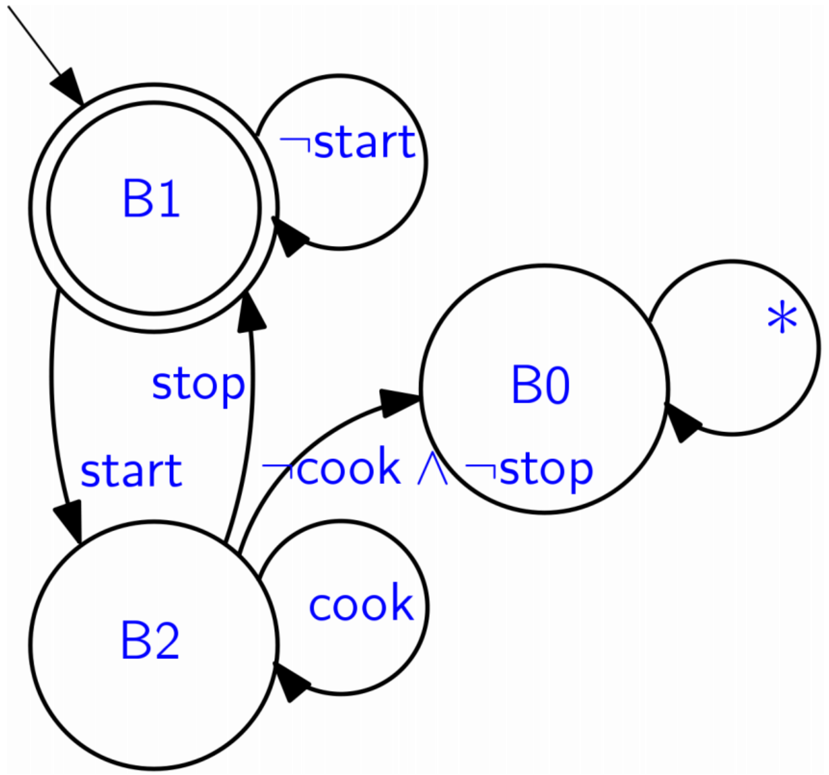
Corner cases:

- which events satisfy ¬start?
- what happens if neither cook nor stop occur in B2 (e.g., start occurs)?

LTL2FSA: complete the transitions

$$\Box\ (\ start \Rightarrow X\ (cook\ U\ stop)\ )$$

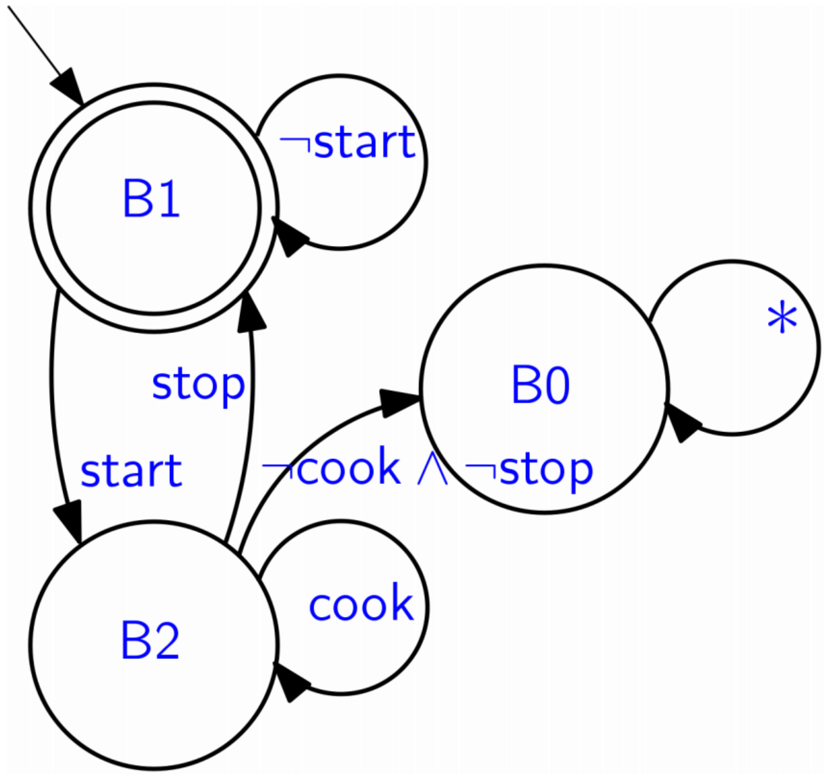
- Always:
  - when start occurs:
    - stop will occur in the future and
    - cook holds until the occurrence of stop
  - if this doesn't happen, fail



LTL2FSA: complement (if deterministic)

[] (start ⇒ X (cook U stop))

- Always:
  - when start occurs:
    - stop will occur in the future and
    - cook holds until the occurrence of stop
  - if this doesn't happen, fail

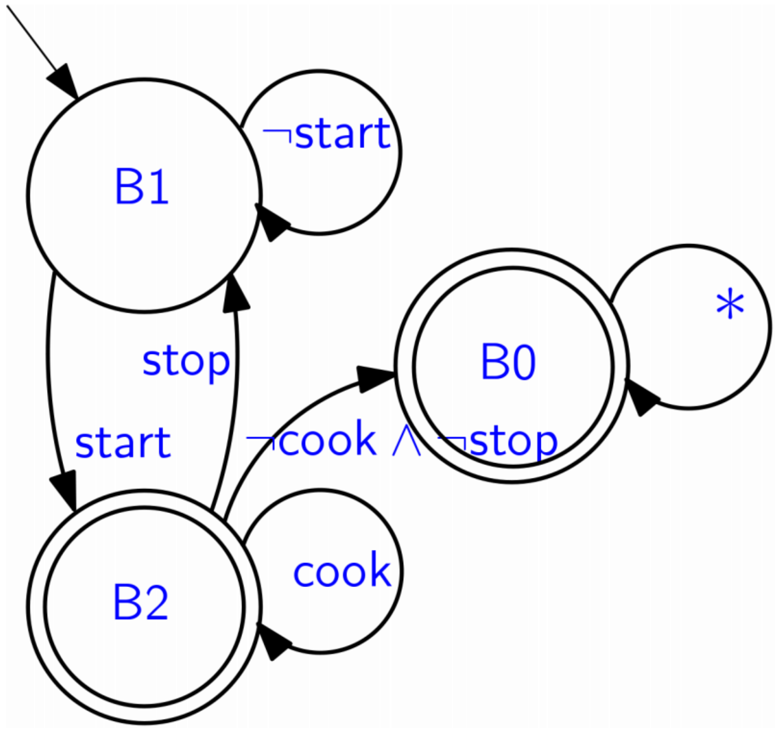


¬[] (start ⇒ X (cook U stop))

≡

<> (start ∧ X (¬cook R ¬stop))

- Sometimes:
  - start occurs and from that moment on:
  - cook becomes false no later than stop

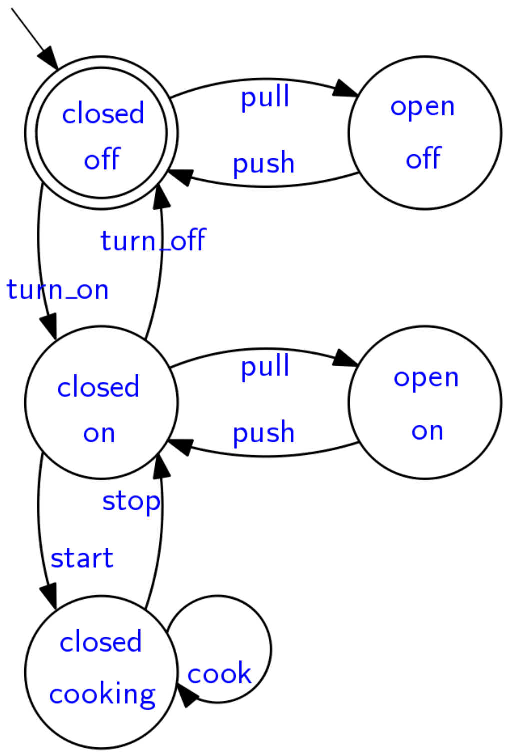


FSA-Intersection: running FSA in parallel

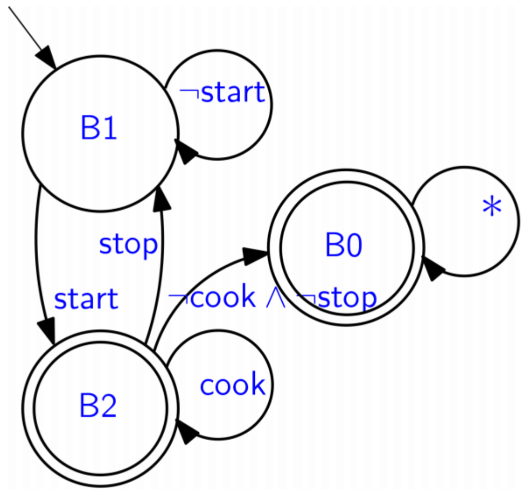
Given automata A, B it is always possible to build automatically an FSA C that accepts precisely the words that both A and B accept.

Automaton C represents all possible parallel runs of A and B where a word is accepted if and only if both A and B accept it. The (simple) construction is called "product automaton".

FSA-Intersection: running FSA in parallel

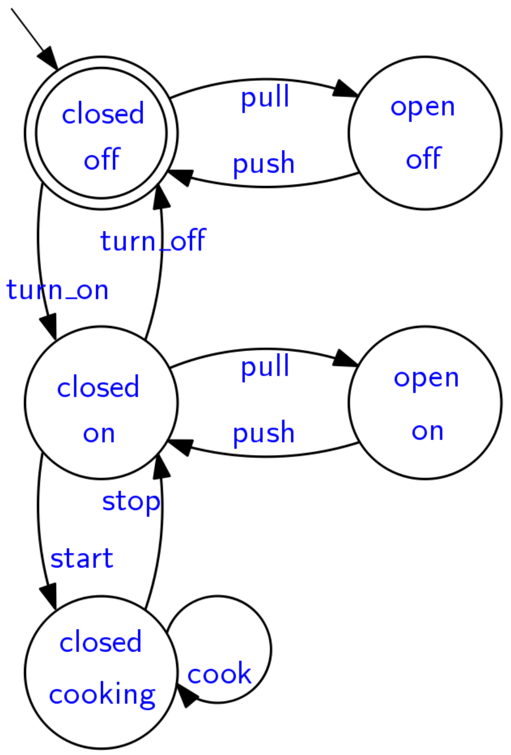


x

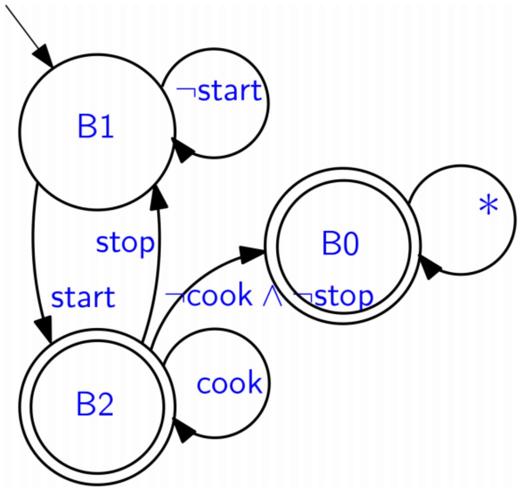


=

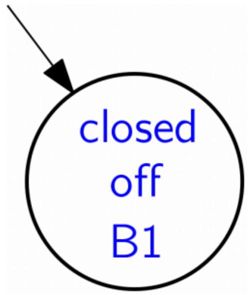
FSA-Intersection: running FSA in parallel



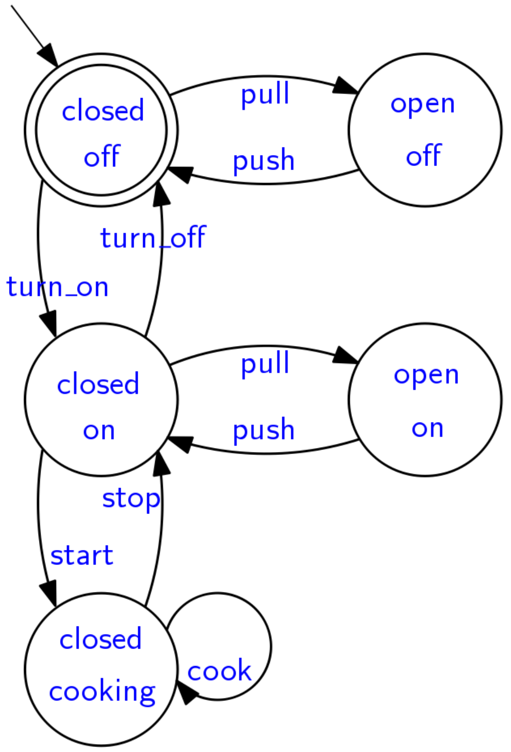
×



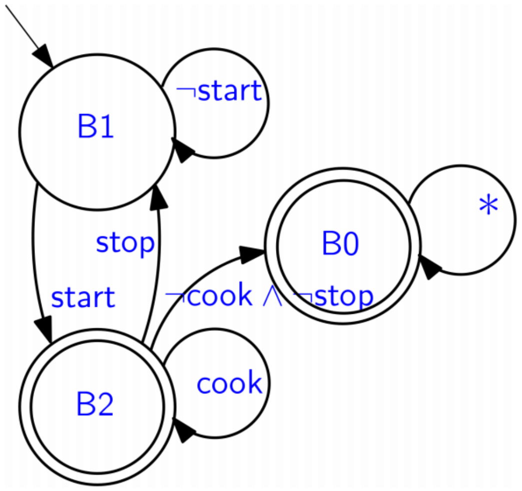
=



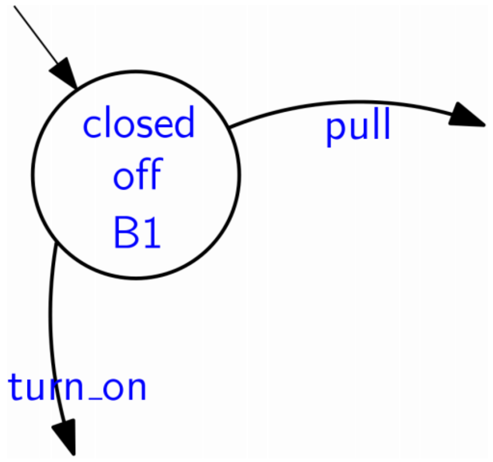
FSA-Intersection: running FSA in parallel



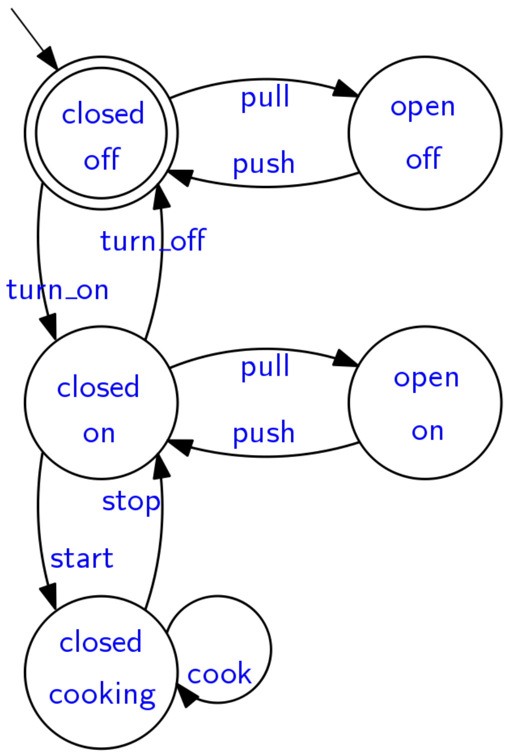
X

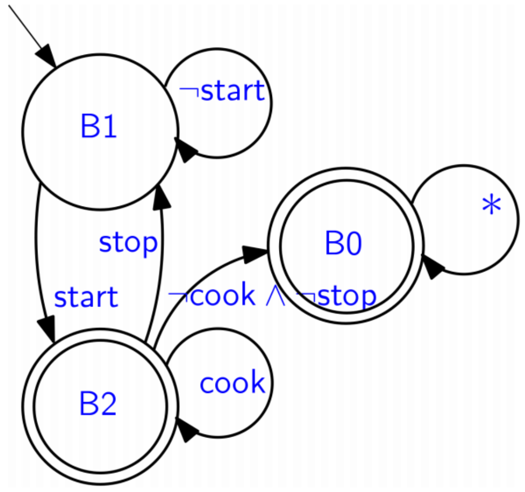


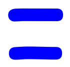
=

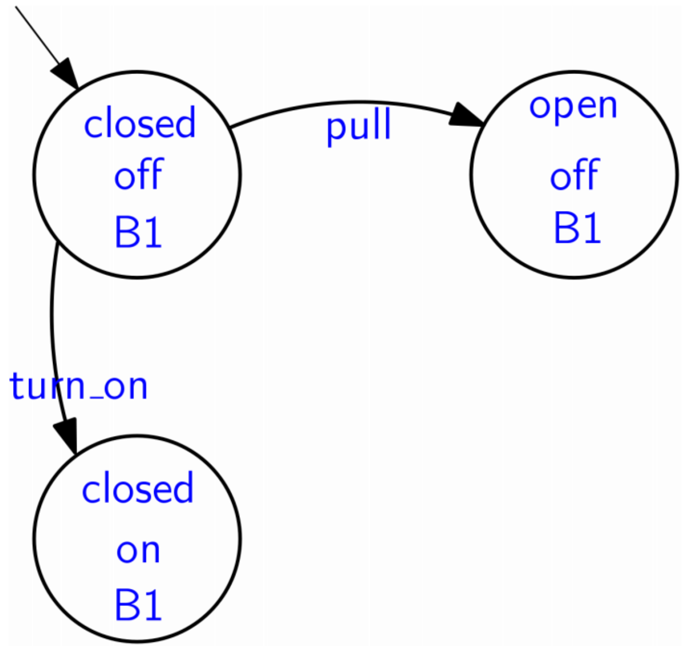


FSA-Intersection: running FSA in parallel

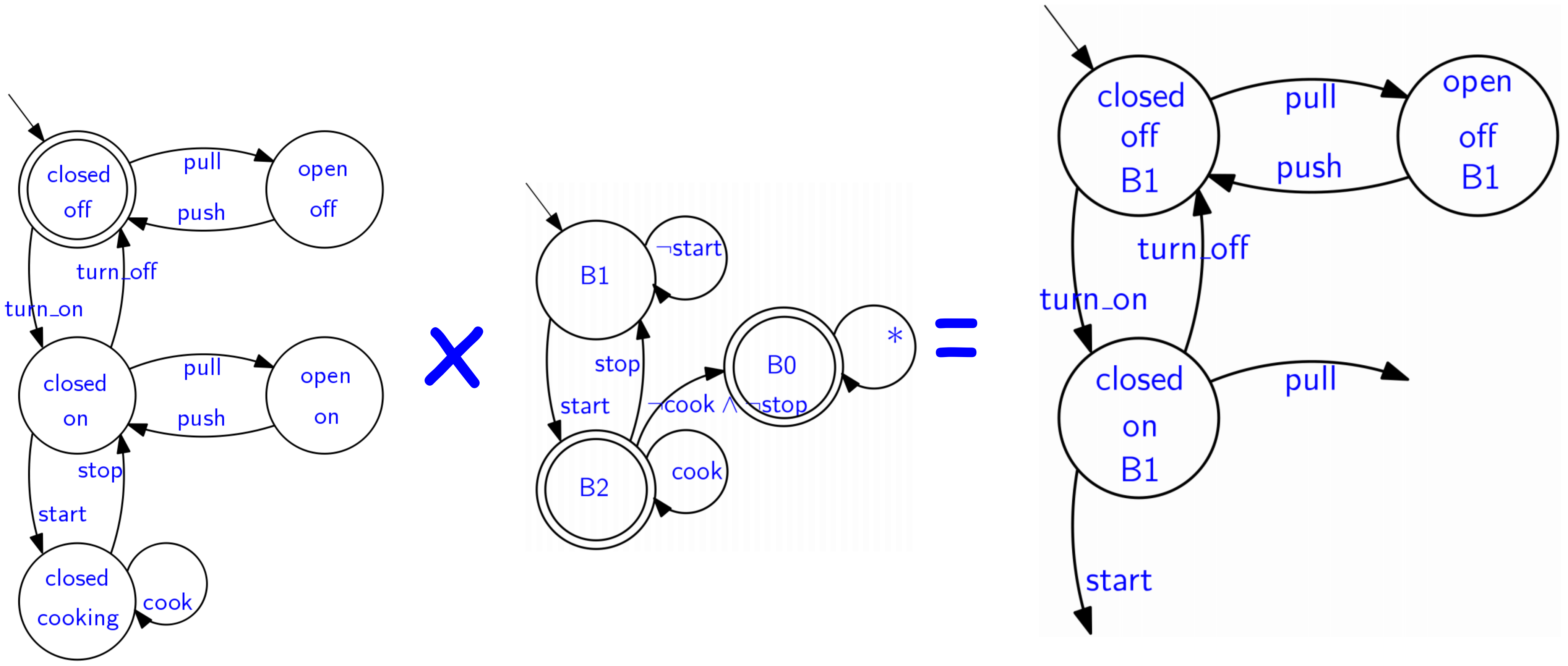




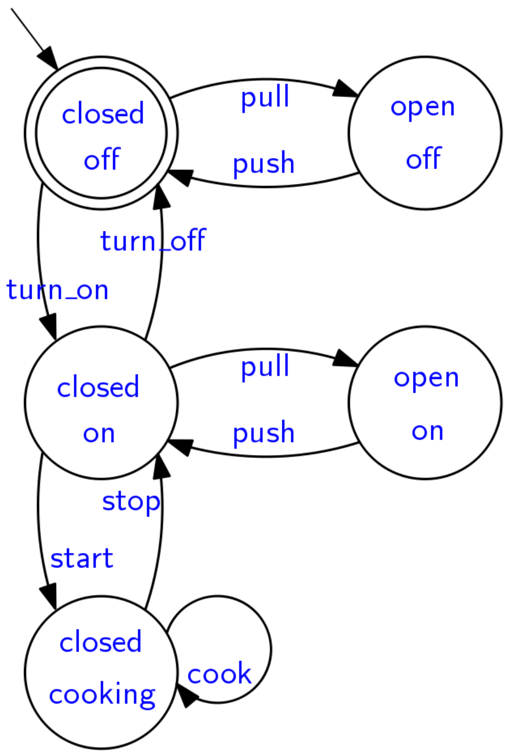




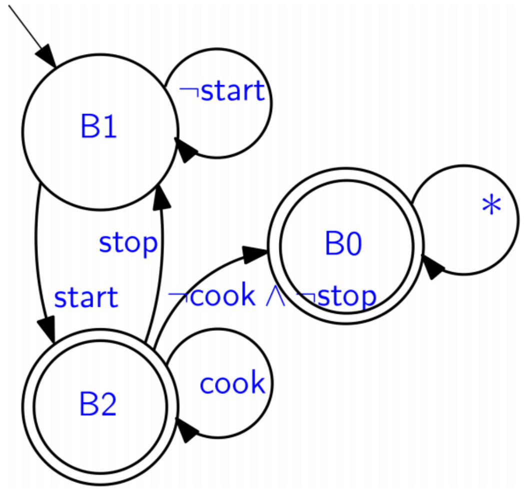
FSA-Intersection: running FSA in parallel



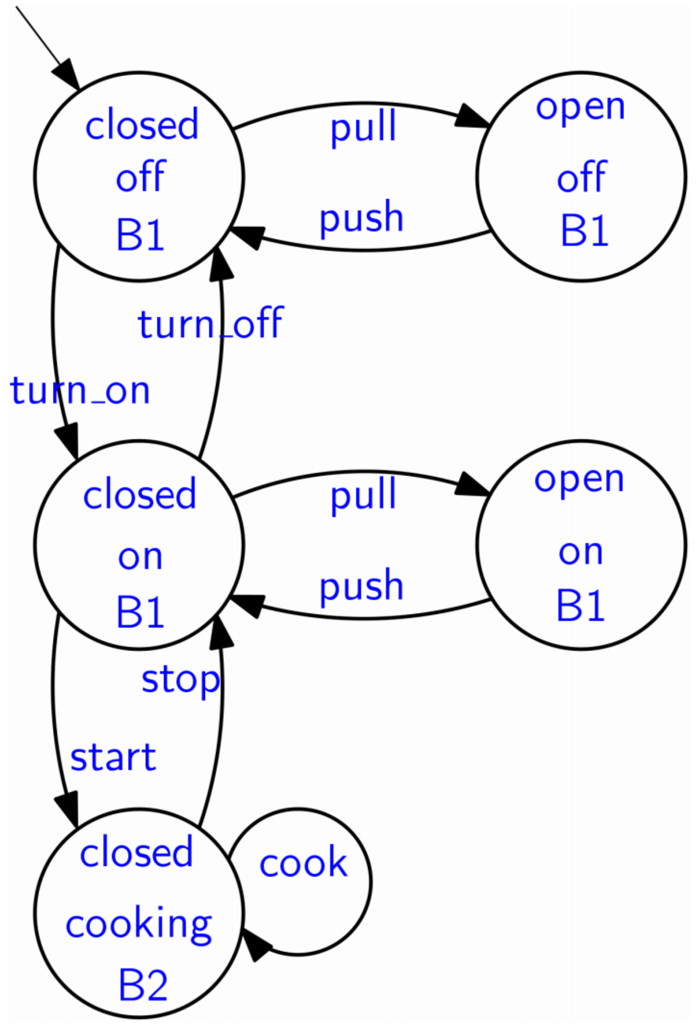
FSA-Intersection: running FSA in parallel



×



=



FSA-Intersection: running FSA in parallel

Def. Given FSA $A=[\Sigma, S^A, I^A, \rho^A, F^A]$ and $B=[\Sigma, S^B, I^B, \rho^B, F^B]$ let $C \triangleq A \times B \triangleq [\Sigma^C, S^C, I^C, \rho^C, F^C]$ be defined as:

- $\Sigma^C \triangleq \Sigma$
- $S^C \triangleq S^A \times S^B$
- $I^C \triangleq \{ (s, t) \mid s \in I^A \text{ and } t \in I^B \}$
- $\rho^C((s, t), \sigma) \triangleq \{ (s', t') \mid s' \in \rho^A(s, \sigma) \text{ and } t' \in \rho^B(t, \sigma) \}$
- $F^C \triangleq \{ (s, t) \mid s \in F^A \text{ and } t \in F^B \}$

Theorem.

$$\langle A \times B \rangle = \langle A \rangle \cap \langle B \rangle$$

FSA-Emptiness: node reachability

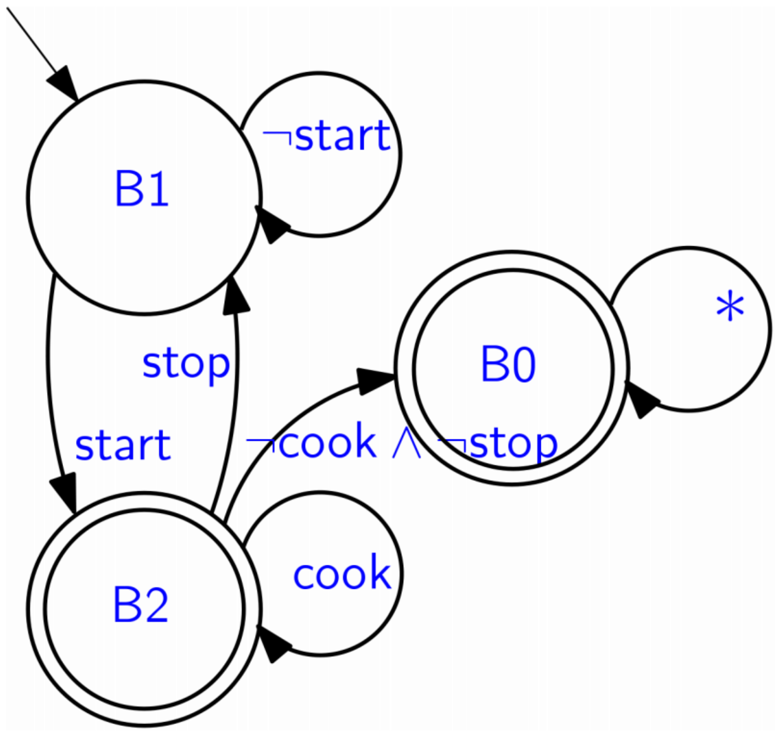
Given an automaton *A* it is always possible to check automatically if it accepts some word.

It suffices to check whether any final state can be reached starting from any initial state.

This amount to checking reachability on the graph representing the automaton: if a path is found, it corresponds to an accepted word; otherwise the automaton accepts an empty language.

FSA-Emptiness: node reachability

It suffices to check whether any final state can be reached starting from any initial state.



From the initial state B1 both accepting states can be reached.

Correspondingly we find the accepted words:

- start
- start cook cook
- start stop start
- ...

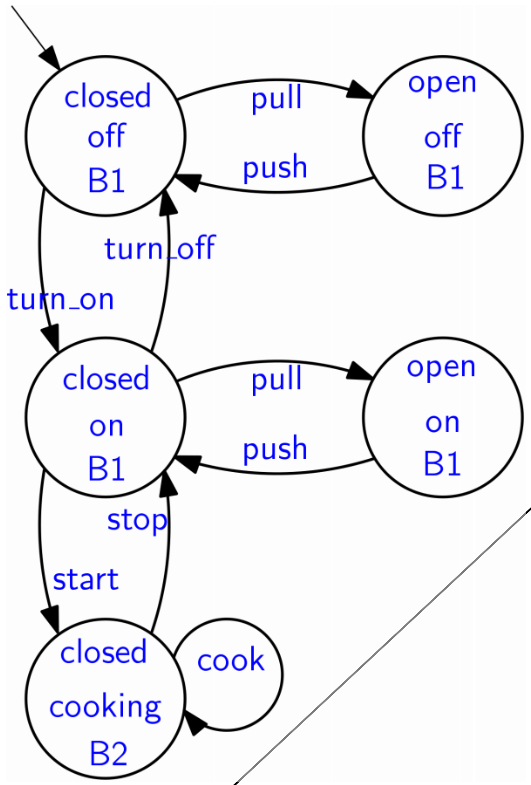
The accepted language is not empty.

Automata-theoretic Model Checking

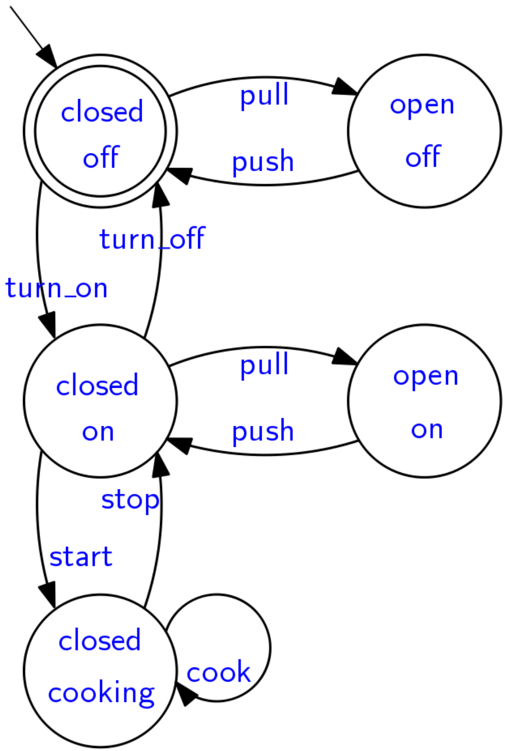
Automata-theoretic Model Checking Algorithm:

- Given: a finite-state automaton A and a temporal-logic formula F
  - TL2FSA: build "tableau" automaton a(¬F)
  - FSA-Intersection: build "product" automaton A x a(¬F)
  - FSA-Emptiness: check whether A x a(¬F) = ∅
- If A x a(¬F) = ∅ then any run of A satisfies F
- If A x a(¬F) ≠ ∅ then show a run of A where F does not hold

Automata-theoretic Model Checking



doesn't accept anything, hence
we have verified:



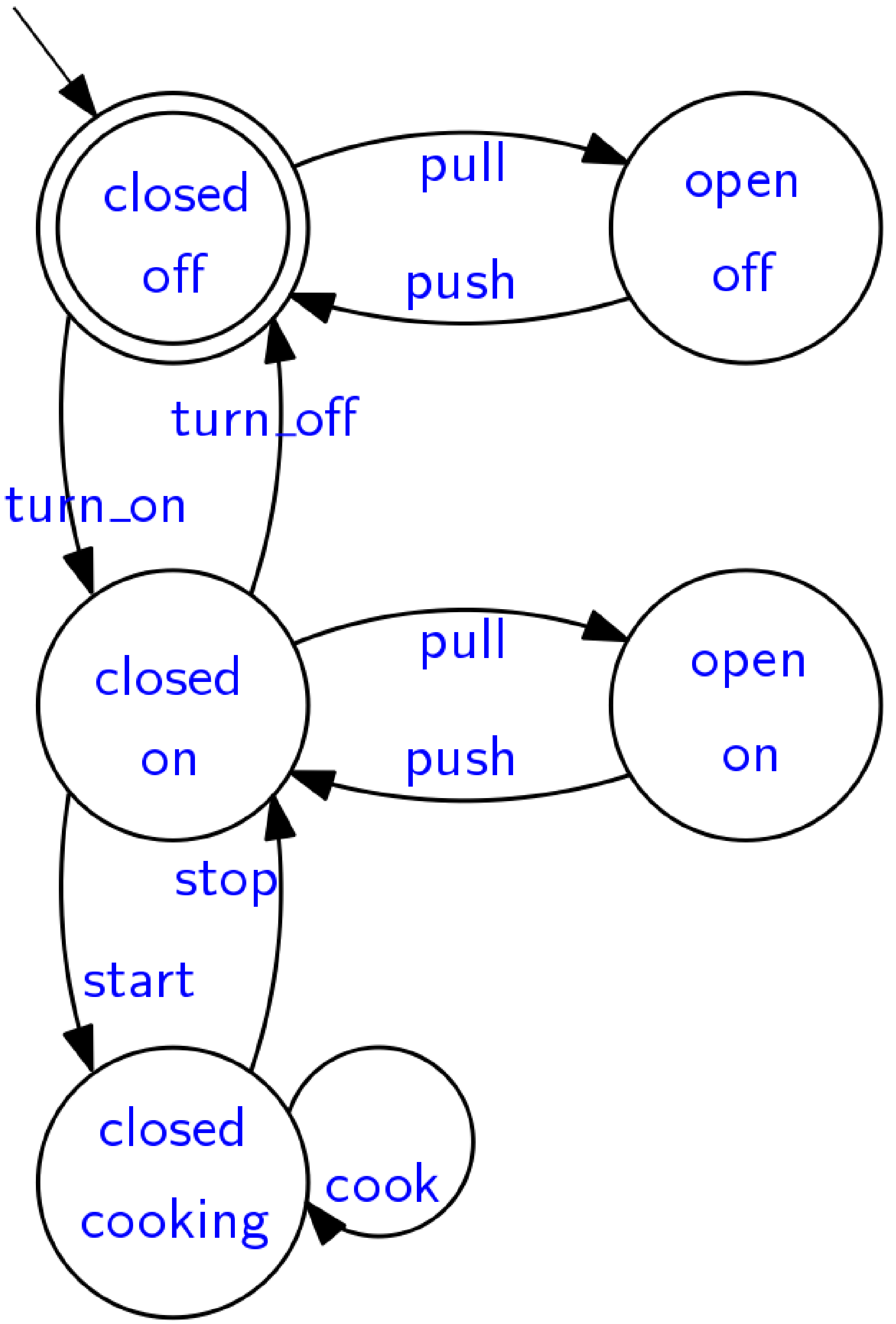
$\models$ [] (start $\Rightarrow$ X (cook U stop))

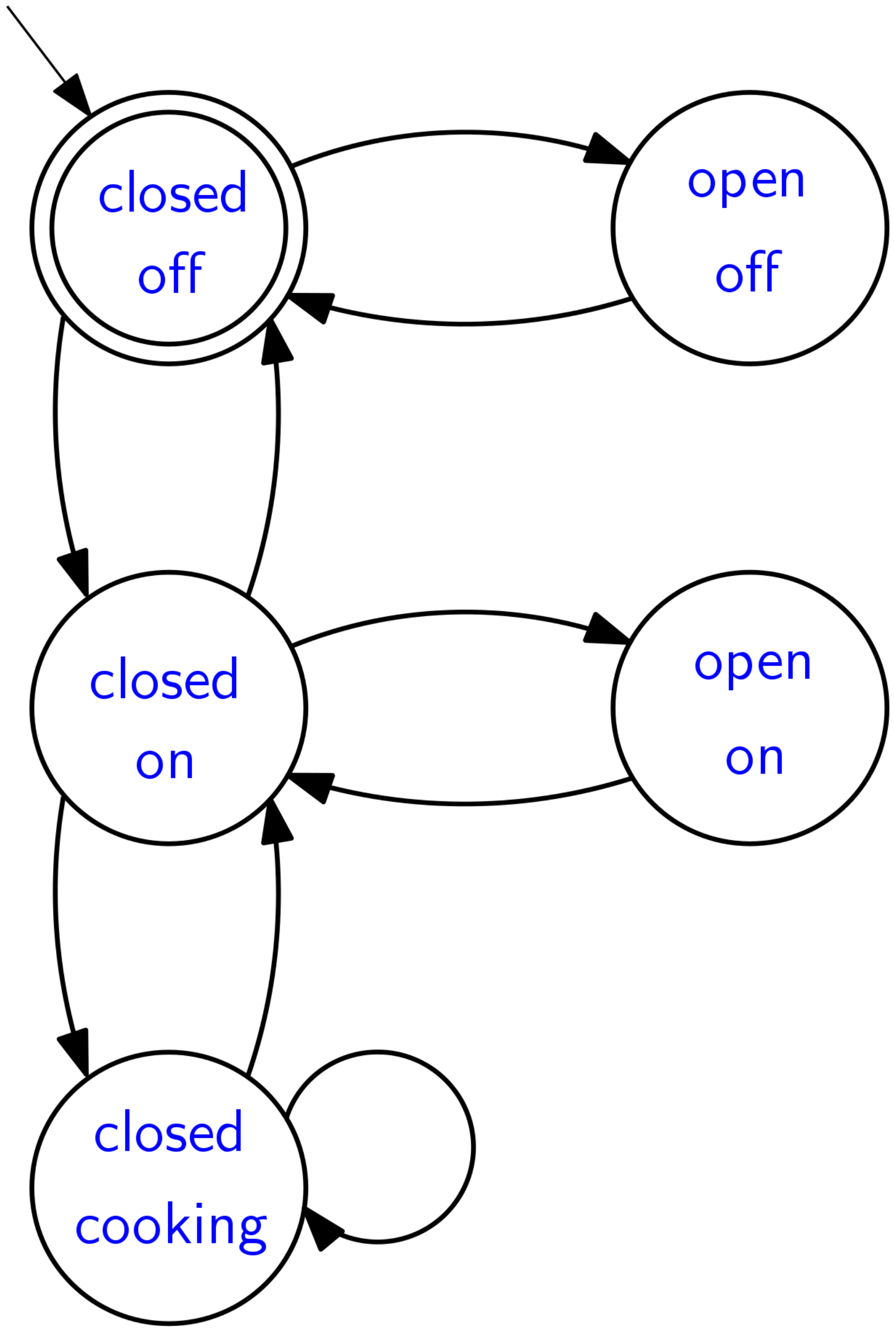
Transition Systems vs. Finite State Automata

Transition Systems

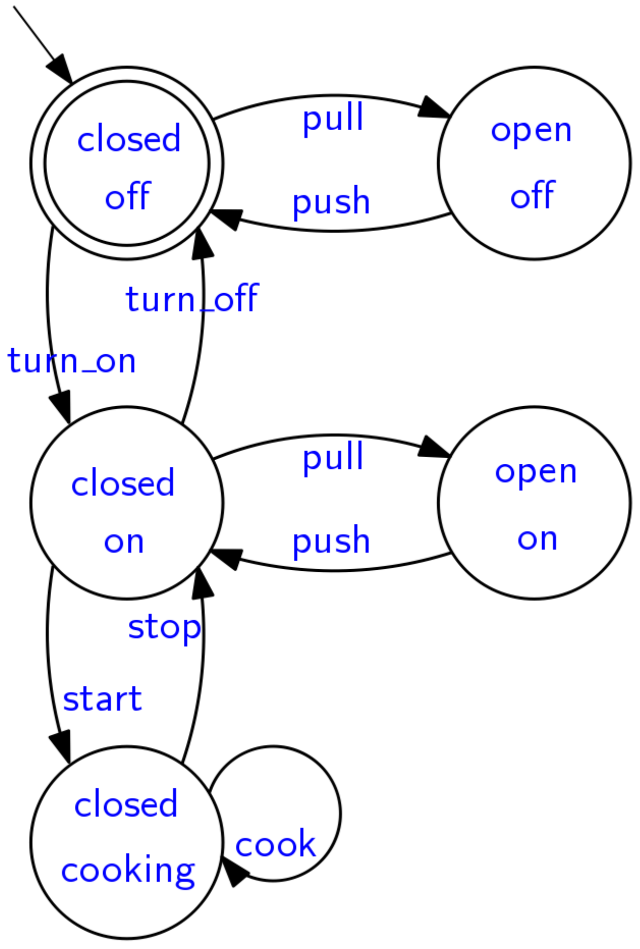
- A slight variant of the model-checking framework uses finite-state transition systems instead of finite-state automata to model the finite-state program/system.
  - Kripke structures is another name for finite-state transition systems.
- A finite-state transition system is a finite-state automaton where propositions are associated to states rather than transition.
- The finite-state transition system and finite-state automaton models are essentially equivalent and it is easy to switch from one to the other.
- The finite-state transition system model is closer to the notion of finite-state program, but the automaton model is more amenable to variants and generalizations (see e.g., class on real-time model-checking).

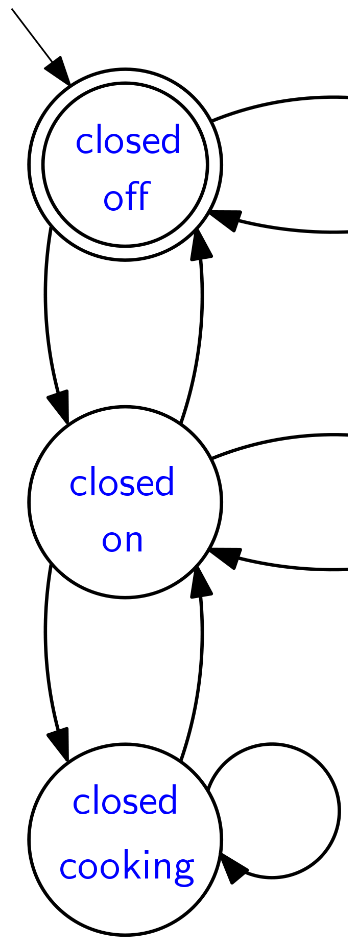
Automaton vs. Transition System





Automaton vs. Transition System

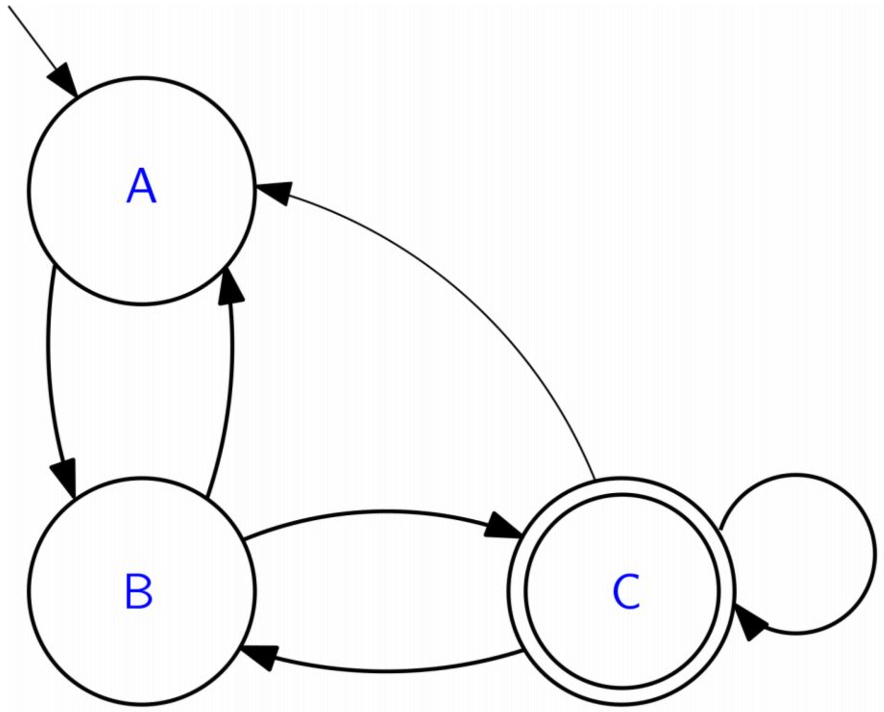


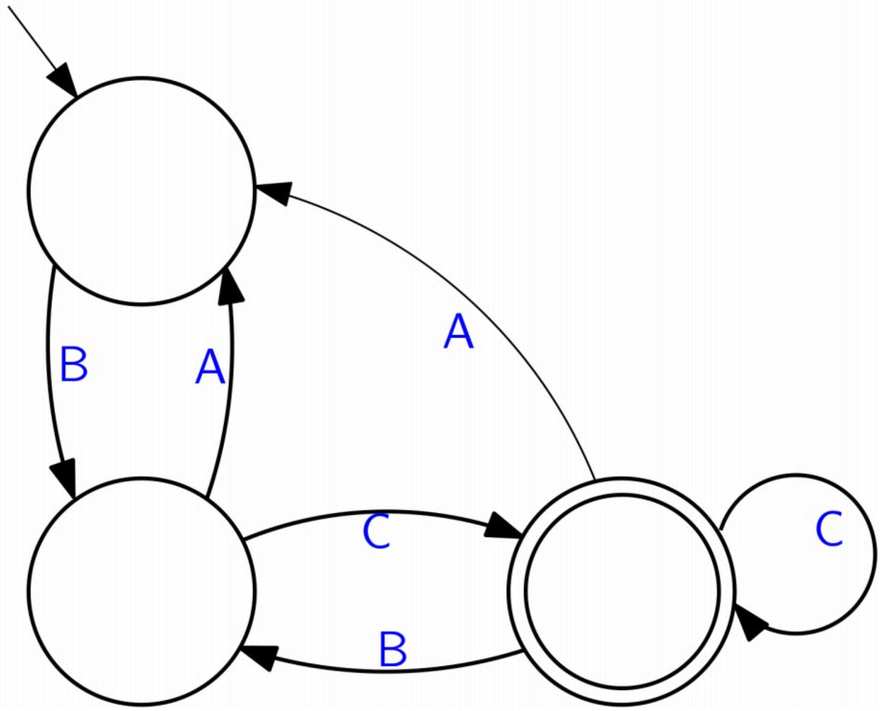


[] (start ⇒ X (cook U stop))

[] (closed-cooking ⇒
X (closed-cooking U closed-on))

Transition System vs. Automaton



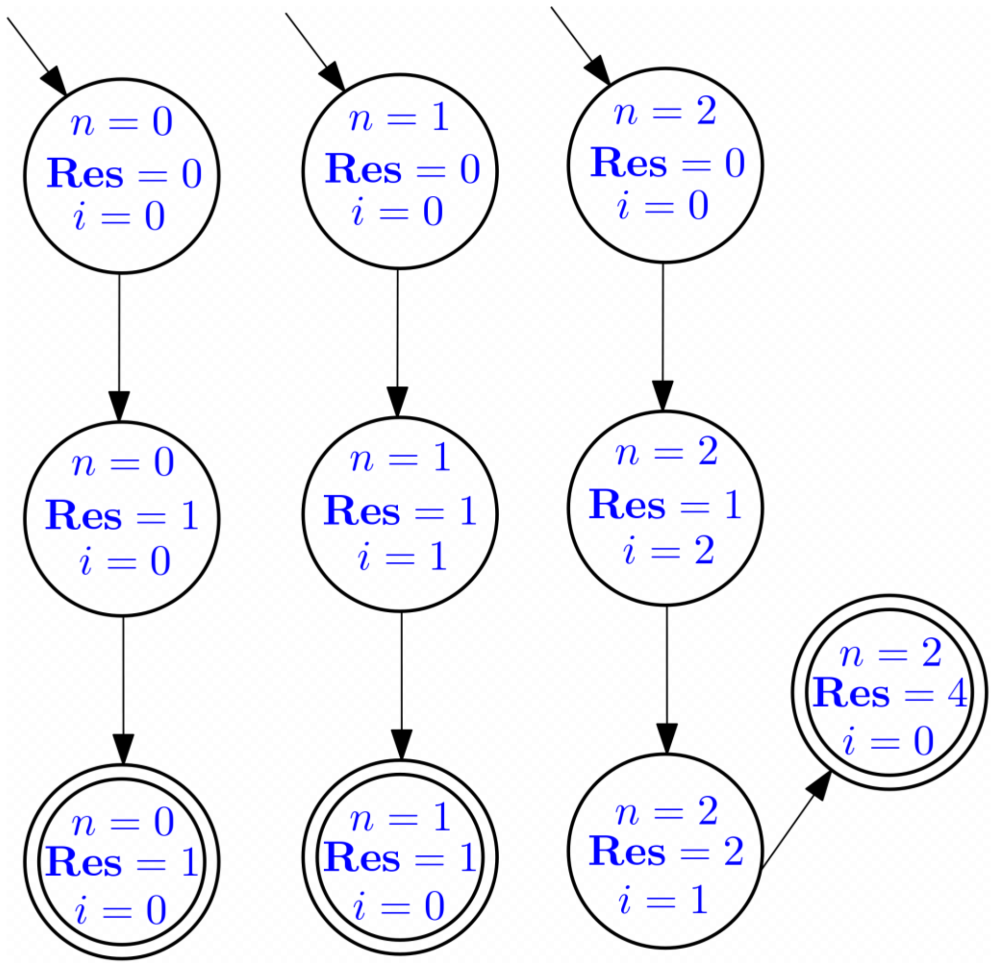


<>[] C

<>[] C

From Programs to Transition Systems

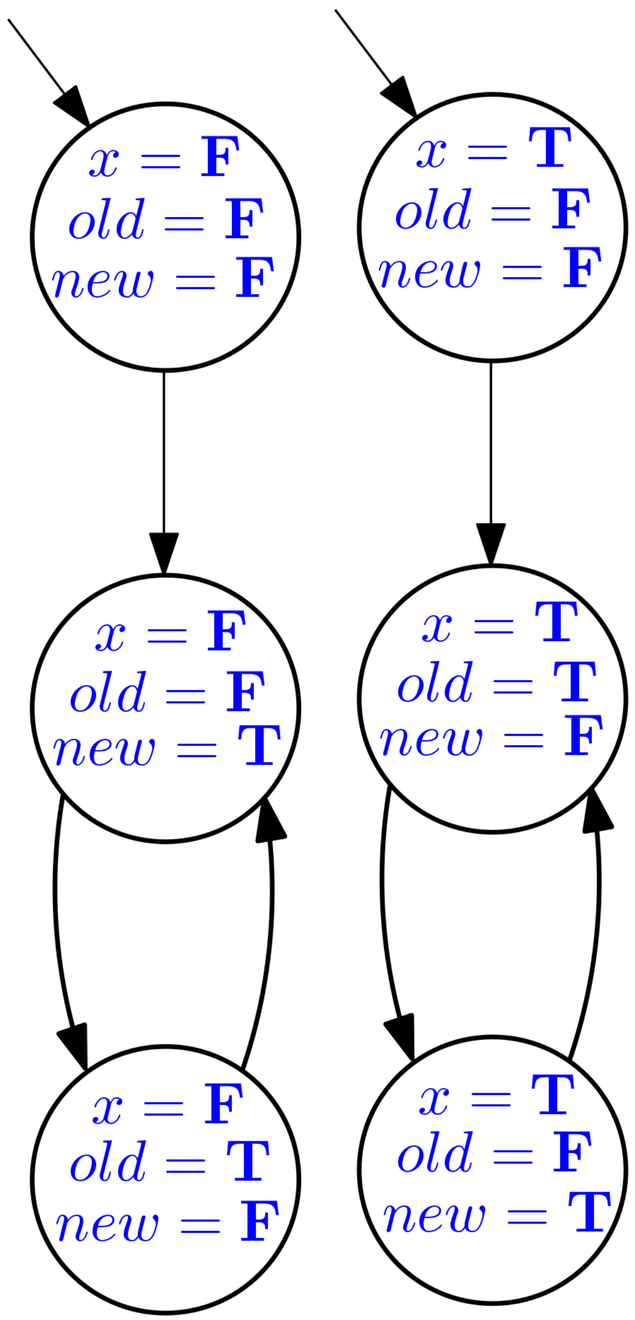
```
n_to_n (n: INTEGER): INTEGER
require 0 ≤ n ≤ 2
local i: INTEGER
do
        from i := n ; Result := 1
        until i = 0
        loop
                Result := Result * n
                i := i - 1
        end
ensure Result = nⁿ end
```



$n = 0$, $\mathbf{Res} = 0$, $i = 0$
$n = 1$, $\mathbf{Res} = 0$, $i = 0$
$n = 2$, $\mathbf{Res} = 0$, $i = 0$
$n = 0$, $\mathbf{Res} = 1$, $i = 0$
$n = 1$, $\mathbf{Res} = 1$, $i = 1$
$n = 2$, $\mathbf{Res} = 1$, $i = 2$
$n = 2$, $\mathbf{Res} = 4$, $i = 0$
$n = 0$, $\mathbf{Res} = 1$, $i = 0$
$n = 1$, $\mathbf{Res} = 1$, $i = 0$
$n = 2$, $\mathbf{Res} = 2$, $i = 1$

From Programs to Transition Systems

```
forever (b: BOOLEAN)
local old, new: BOOLEAN
do
        from old := b ; new := not b
        until old = new
        loop
                old := new
                new := not old
        end
end
```


$x = \mathbf{F}$ $old = \mathbf{F}$ $new = \mathbf{F}$
$x = \mathbf{T}$ $old = \mathbf{F}$ $new = \mathbf{F}$
$x = \mathbf{F}$ $old = \mathbf{F}$ $new = \mathbf{T}$
$x = \mathbf{T}$ $old = \mathbf{T}$ $new = \mathbf{F}$
$x = \mathbf{F}$ $old = \mathbf{T}$ $new = \mathbf{F}$
$x = \mathbf{T}$ $old = \mathbf{F}$ $new = \mathbf{T}$

Variants of the Model-Checking Algorithm

Variants of the Model-Checking Algorithm

The basic model-checking algorithm:

- TL2FSA: build automaton $a(\neg F)$
- FSA-Intersection: build automaton $A \times a(\neg F)$
- FSA-Emptiness: check whether $A \times a(\neg F) = \emptyset$

can be refined into different variants:

- Explicit-state model-checking
- Symbolic (BDD-based) model-checking
- Bounded (SAT-based) model-checking

The variants differ in how they represent automata and formulae and how they analyze them. Hybrid approaches are also possible.

Explicit-state Model Checking

Explicit-state model-checking represents automata explicitly as graphs:

- TL2FSA: build automaton $a(\neg F)$
  - the automaton is represented as a graph
- FSA-Intersection: build automaton $A \times a(\neg F)$
  - the intersection is usually built on-the-fly while checking emptiness, because the product automaton can be large
- FSA-Emptiness: check whether $A \times a(\neg F) = \emptyset$
  - a search on the expanded intersection graph looks for reachable accepting nodes

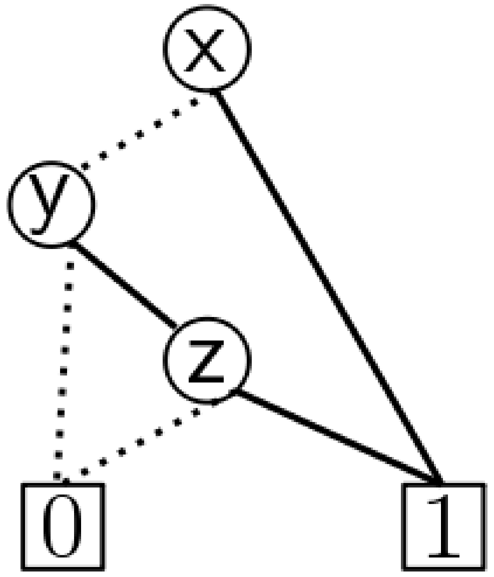
SPIN is an example of explicit-state model checker.

Symbolic Model Checking

Symbolic model-checking represents automata implicitly (symbolically) through their transition functions encoded as BDDs (Binary Decision Diagrams):

- A BDD is an efficient representation of Boolean functions (i.e., truth tables) as acyclic graphs
- Logic operations (e.g., conjunction, negation) can be performed efficiently directly on BDDs

$x \vee (y \wedge z)$

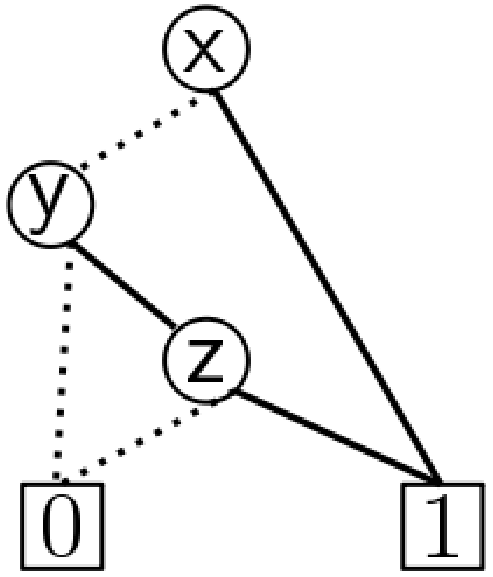


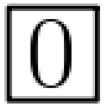
Symbolic Model Checking

Logic operations (e.g., conjunction, negation) can be performed efficiently directly on BDDs

- TL2FSA: build automaton a(¬F)
  - the transition function of the automaton is represented as a BDD
- FSA-Intersection: build automaton A x a(¬F)
  - the intersection is a BDD built by manipulating the two BDDs
- FSA-Emptiness: check whether A x a(¬F) = ∅
  - emptiness checking is also performed directly on the BDD
    - it amount to reduction to a canonical form and then comparison with the canonical BDD for unsatisfiable Boolean functions

$x \vee (y \wedge z)$





SMV is an example of symbolic model checker.

Bounded Model Checking

Bounded model-checking considers all paths of bounded size on the automaton and represents them as a propositional formula. Propositional formulas are then checked for satisfiability with SAT-solvers (i.e., automatic provers for propositional satisfiability).

- The bound k of the path size is an additional input to the model-checking problem with respect to standard model-checking. However, if the bound is “large enough” the problem is equivalent to standard model-checking.
- Even if the encoding as a propositional formula is quite large, SAT-solvers can handle huge (e.g., > 10^5 propositions) formulas efficiently.

verification tool
NP-completeness should never scare the ~~compiler~~ writer.

-- *Andrew W. Appel*

Bounded Model Checking

- TL2FSA: build automaton $a(\neg F)$
  - the LTL formula is translated directly into a propositional formula $p(\neg F)$
- FSA-Intersection: build automaton $A \times a(\neg F)$
  - the product of two propositional formulas is simply their conjunction $p(A) \wedge p(\neg F)$
- FSA-Emptiness: check whether $A \times a(\neg F) = \emptyset$
  - emptiness checking is equivalent to satisfiability checking of $p(A) \wedge p(\neg F)$

nuSMV and Zot are examples of bounded model checkers.

Variants of the Model-Checking Approach

Variants of the Model-Checking Problem

The Model Checking problem:

- Given: a finite-state automaton A and a temporal-logic formula F
- Determine: if any run of A satisfies F or not
  - if not, also provide a counterexample: a run of A where F does not hold

The general problem can be refined into variants, according to the nature of A and F.

- The same generic automata-theoretic solution

  (TL2FSA -> Intersection -> Emptiness)

  applies to any of these variants

  (modulo some technicalities)

Variants of the Model-Checking Problem

The general problem can be refined into variants, according to the nature of A and F.

Classes of automata:

- Finite State Automata (FSA)
- Büchi Automata (BA)
- Alternating Automata (AA)
- ...

Classes are not disjoint

Classes of temporal logic:

- Linear-time temporal logic
- Branching-time temporal logic
- Temporal logic with past operators
- ...

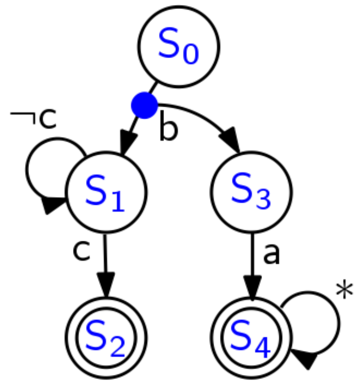
Classes are not disjoint

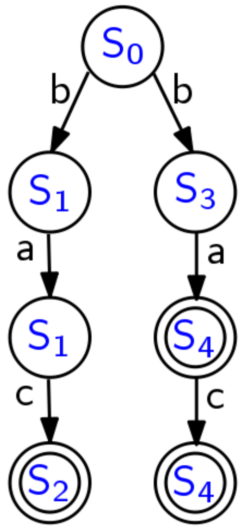
Automata Classes

- Finite-state Automata (FSA)
  - those presented in this lecture
  - FSA runs correspond to finite words (words of finite length)
- Büchi Automata (BA)
  - named after Julius Büchi (Swiss logician, ETH graduate)
  - BA runs correspond to infinite words (words of unbounded length)
    - this complicates the definitions of acceptance, product, and complement, as well as the algorithm for emptiness
  - infinite words are needed to model:
    - reactive systems: ongoing interaction with environment
      - e.g., control system, interactive protocol, etc.
    - liveness and fairness
      - e.g., "process P will not starve"
  - the most common presentation of linear-time model-checking uses BA

Automata Classes (cont'd)

- Alternating Automata (AA)
  - Alternation is a generalization of nondeterminism to universality:
    - existential nondeterminism: when multiple parallel runs are possible accept iff at least one of them is accepting
    - universal nondeterminism: when multiple parallel runs are possible accept iff all of them are accepting
  - AA runs correspond to trees (of finite or infinite height)
    - a tree represents parallel runs over the same input word
      - e.g.: an AA accepting ba(a|b)*c and a run on word "bac"





  - AA are also used as intermediate representation in the translation from LTL to BA

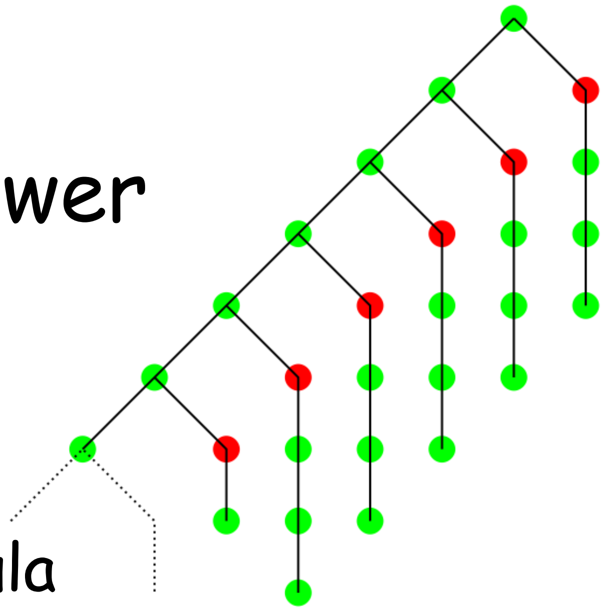
Temporal Logic Classes

- Linear-time Temporal Logic (LTL)
  - the one presented in this lecture
  - LTL formulae express properties of linear sequences, that is words
    - linear: every element has only one possible successor
    - linear time: every step has only one possible "future"
- Branching-time Temporal Logic
  - includes path quantifiers in the syntax
  - for example CTL (Computation Tree Logic):
    $F ::= p \mid \neg F \mid F \wedge G \mid \exists X\, F \mid \forall X\, F \mid F\, \exists U\, G \mid F\, \forall U\, G$
  - branching-time formulae express properties of branching structures, that is trees
    - branching: an element can have multiple possible successors
    - branching time: a step can have many possible "futures"
  - e.g.: $\exists$<> p: "there exists a path where p eventually holds"

Linear vs. Branching

LTL and CTL have different strengths and weaknesses

- Expressiveness: LTL and CTL have incomparable expressive power
  - CTL formula ∀<>∀[] p: "p will stabilize at True within a bounded amount of time" doesn't have an equivalent LTL formula
  - LTL formula <>[] p: "p is ultimately True in every computation" doesn't have an equivalent CTL formula
  - see infinite computation tree, where LTL formula <>[] p holds but CTL formula ∀<>∀[] p doesn't. (p holds precisely in green nodes)



Linear vs. Branching

LTL and CTL have different strengths and weaknesses

- Complexity: (checking whether $A \models F$)
  - CTL model-checking: $O(|A| \cdot |F|)$
  - LTL model-checking: $O(|A| \cdot 2^{|F|})$ and PSPACE-complete
  - However: There is life after exponential explosion -- *Moshe Vardi*
    - $|F|$ usually much smaller than $|A|$
    - CTL advantage vanishes when model-checking open systems
    - In practice similar performances with formulas that are expressible in both logics
- Usability and intuitiveness:
  - CTL quite unintuitive
  - LTL intuitive but cannot express some interesting properties (beyond CTL ones)

Temporal Logic Classes (cont'd)

- It is possible to add past temporal operators to temporal logics
- Typically done with LTL giving LTL+P:
  - Y F: "yesterday F occurred"
  - F S G: "F holds since G"
  - <> F: "F held sometime in the past"
  - ...

Temporal Logic Classes (cont'd)

- Past operators do not increase the expressive power of LTL: everything that can be expressed with LTL+P can also be expressed in LTL (without past operators)
- Past operators increase the usability of LTL
  - "Every alarm is due to a fault"
    - with past operators:

      [] (alarm ⇒ <>fault)
    - without past operators:

      ¬ (¬fault U (alarm ∧ ¬fault))

A Brief History of Model Checking

Basic ingredients:

- Kripke structures
  - Kripke, circa 1963
- Büchi automata
  - Büchi, 1960
- Temporal ("tense") logic
  - Prior, 1957
  - Kamp, 1968

Into computer science:

- Using temporal logic to reason about programs
  - Pnueli, 1977
- Model checking
  - Clarke & Emerson, 1981
  - Queille & Sifakis, 1981
- Automata-theoretic framework
  - Vardi & Wolper, circa 1986
- Implementations
  - SPIN, circa 1990
  - SMV, circa 1990
- Many extensions...

Everything's a Model-Checker

- Model-checking techniques have gained much popularity, both in the research community and among practitioners
  - 2007 ACM Turing award to Clarke, Emerson, and Sifakis for the invention of Model Checking
  - Hardware industry (e.g., Intel) uses model-checking techniques for production hardware
- The model-checking framework has been modified and extended in many different directions
  - real-time and hybrid model-checking (see future class)
  - probabilistic model-checking
  - software model-checking (see future class)
    - abstraction & refinement
  - infinite-state model-checking
  - Petri net model-checking
  - ...

Everything's a Model-Checker

- Some extensions are so far-away from the original technique that "model-checking" is almost misnomer for them
- However, the popularity of model checking has also loosened the meaning of the term, so that sometimes "model checking" is synonym with "algorithmic (automated) verification"
  - From an historic point of view, it is essentially true that model checking has been the first workable technique for automated verification