Automata-Theoretic LTL Model-Checking

Arie Gurfinkel
arie@cmu.edu

SEI/CMU
Determines Patterns on Infinite Traces

Atomic Propositions

Boolean Operations

Temporal operators

- $a$ → “a is true now”
- $Xa$ → “a is true in the next state”
- $Fa$ → “a will be true in the future”
- $Ga$ → “a will be globally true in the future”
- $a U b$ → “a will hold true until b becomes true”
LTL - Linear Time Logic (Pn 77)

Determines Patterns on Infinite Traces

Atomic Propositions
Boolean Operations
Temporal operators

\( a \)  
“a is true now”

\( X a \)  
“a is true in the next state”

\( Fa \)  
“a will be true in the Future”

\( Ga \)  
“a will be Globally true in the future”

\( a \cup b \)  
“a will hold true Until b becomes true”
Determines Patterns on Infinite Traces

Atomic Propositions
Boolean Operations
Temporal operators

\( a \)  “a is true now”
\( X a \)  “a is true in the,next state”
\( Fa \)  “a will be true in the,Future”
\( Ga \)  “a will be, Globally true in the future”
\( a \, U \, b \)  “a will hold true Until b becomes true”
LTL - Linear Time Logic (Pn 77)

Determines Patterns on Infinite Traces

Atomic Propositions
Boolean Operations
Temporal operators

\(a\) “a is true now”
\(Xa\) “a is true in the next state”
\(Fa\) “a will be true in the Future”
\(Ga\) “a will be Globally true in the future”
\(a \mathbin{U} b\) “a will hold true Until b becomes true”
LTL - Linear Time Logic (Pn 77)

Determines Patterns on Infinite Traces

Atomic Propositions
Boolean Operations
Temporal operators

\( a \) “a is true now”
\( \text{X} a \) “a is true in the next state”
\( \text{F} a \) “a will be true in the Future”
\( \text{G} a \) “a will be Globally true in the future”
\( a \text{ U } b \) “a will hold true Until b becomes true”
Outline

• Automata-Theoretic Model-Checking
  ◦ Finite Automata and Regular Languages
  ◦ Automata over infinite words: Büchi Automata
  ◦ Representing models and formulas with automata
  ◦ Model checking as language emptiness
Finite Automata

A finite automaton $\mathcal{A}$ (over finite words) is a tuple $(\Sigma, Q, \Delta, Q^0, F)$, where

- $\Sigma$ is a finite alphabet
- $Q$ is a finite set of states
- $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation
- $Q^0 \subseteq Q$ is a set of initial states
- $F \subseteq Q$ is a set of final states
Finite Automaton: An Example

\[ \Sigma = \{a, b, c\}, Q = \{q_0, q_1\}, Q^0 = \{q_0\}, F = \{q_1\} \]
A Run

• A run of $\mathcal{A}$ over a word $v \in \Sigma^*$ of length $|v|$ is a mapping $\rho : \{0, 1, \ldots, |v|\} \rightarrow Q$ s.t.
  ◦ First state is the initial state: $\rho(0) \in Q^0$
  ◦ States are related by transition relation:
    $$\forall 0 \leq i \leq |v| \cdot (\rho(i), v(i), \rho(i + 1)) \in \Delta$$

• A run is a path in $\mathcal{A}$ from $q_0$ to a state $\rho(|v|)$ s.t. the edges are labeled with letters in $v$

• A run is accepting if it ends in an accepting state: $\rho(|v|) \in F$.

• $\mathcal{A}$ accepts $v$ iff exists an accepting run of $\mathcal{A}$ on $v$. 
An Example of a Run

- A run $q_0, q_1, q_1, q_1, q_0$ on $aacb$ is accepting
- A run $q_0, q_0, q_0, q_0, q_0$ on $bbbb$ is accepting
- A run $q_0, q_0, q_1, q_1, q_1$ on $baac$ is rejecting
Language

The language $\mathcal{L}(A) \subseteq \Sigma^*$ is the set of all words in $\Sigma^*$ accepted by $A$.

The language is $\{\epsilon, b, bb, ccc, bab, \ldots\}$

That is, a regular expression: $\epsilon + a(a + c)^*b(b + c)^*$
Regular Languages

- A set of strings is *regular* if it is a language of a finite automaton (i.e., recognizable by a finite automaton).
- An automaton is *deterministic* if the transition relation is deterministic for every letter in the alphabet:

\[ \forall a \cdot (q, a, q') \in \Delta \land (q, a, q'') \in \Delta \Rightarrow q' = q'' \]

otherwise, it is *non-deterministic*.
- NFA = DFA: Every non-deterministic finite automaton (NFA) can be translated into a language-equivalent deterministic automaton (DFA).
Automata on Infinite Words

- Reactive programs execute forever – need infinite sequences of states to model them!
- Solution: finite automata over infinite words.
- Simplest case: Büchi automata
  - Same structure as automata on finite words
    - ... but different notion of acceptance
  - Recognize words from $\Sigma^\omega$ (not $\Sigma^*$!)
    - $\Sigma = \{a, b\}$  $v = abaabaaab...$
    - $\Sigma = \{a, b, c\}$
      $L_1 = \{v \mid \text{in } v \text{ after every } a \text{ there is a } b\}$
      Some words in $L_1$:
      - $ababab\cdots aaabaaab\cdots$
      - $abbabbabb\cdots accbacci\cdots$
Infinite Run and Acceptance

- Recall, $F$ is the set of accepting states.
- A run $\rho$ of a Büchi automaton $A$ is over an infinite word $v \in \Sigma^\omega$. Domain of the run is the set of all natural numbers.
- Let $\text{inf}(\rho)$ be the set of states that appear infinitely often in $\rho$:
  $$\text{inf}(\rho) = \{q | \forall i \in \mathbb{N} \cdot \exists j \geq i \cdot \rho(j) = q\}$$
- A run $\rho$ is accepting (Büchi accepting) iff $\text{inf}(\rho) \cap F \neq \emptyset$.
- A set of strings is $\omega$-regular iff it is recognizable by a Büchi automaton.
Example

\[ q_0 \rightarrow^a q_1 \rightarrow^b q_0 \]

States:
- \( q_0 \) with transitions:
  - \( q_0 \rightarrow^a q_1 \)
  - \( q_1 \rightarrow^b q_0 \)

Transitions:
- \( q_0 \rightarrow^b q_0 \)
- \( q_1 \rightarrow^a q_1 \)

Input symbols:
- \( a \)
- \( b \)
- \( c \)
Example

Language of the automaton is: \( ((b + c)^\omega a(a + c)^*b)^\omega \)

This is an \( \omega \)-regular expression
Examples

Let $\Sigma = \{0, 1\}$. Define Büchi automata for the following languages:

1. $L = \{v \mid 0 \text{ occurs in } v \text{ exactly once}\}$
2. $L = \{v \mid \text{after each 0 in } v \text{ there is a 1}\}$
3. $L = \{v \mid v \text{ contains finitely many 1’s}\}$
4. $L = (01)^n \Sigma^\omega$
5. $L = \{v \mid 0 \text{ occurs in every even position of } v\}$
Closure Properties

Büchi-recognizable languages are closed under . . .

• (alphabet) projection and union
  ◦ Same algorithms as Finite Automata

• intersection
  ◦ Different construction from Finite Automata

• complement
  ◦ i.e., from a Büchi automaton $A$ recognizing $L$ one can construct an automaton $\overline{A}$ recognizing $\Sigma^\omega - L$.
  ◦ $\overline{A}$ has order of $O\left(2^{Q \log Q}\right)$ states, where $Q$ are states in $A$ [Safra’s construction]
Complementation: Example

Complement is easy for deterministic Büchi automata:

\[ A \]

\[ b, c \]

\[ q_0 \]

\[ a \]

\[ a, c \]

\[ q_1 \]

\[ b \]
Complementation: Example

Complement is easy for deterministic Büchi automata:

\[ A \]

\[ q_0 \quad a \quad b \quad q_1 \]

\[ \overline{A} \]

\[ a, b, c \quad a \quad a, c \]
Complementation: Example

Complement is easy for deterministic Büchi automata:

\[ A \]

But, Büchi automata are not closed under determinization!!!
Intersection (Special Case)

Büchi automata are closed under intersection [Chouka74]:

- given two Büchi automata (note all states of $B_1$ are accepting):

  \[ B_1 = (\Sigma, Q_1, \Delta_1, Q_1^0, Q_1) \quad B_2 = (\Sigma, Q_2, \Delta_2, Q_2^0, F_2) \]

- Define $B_\cap = (\Sigma, Q_1 \times Q_2, \Delta', Q_1^0 \times Q_2^0, Q_1 \times F_2)$, where
  \[ ((s_1, s_2), a, (s_1', s_2')) \in \Delta' \iff (s_i, a, s_i') \in \Delta_i, i = 1, 2 \]

- Then, \( \mathcal{L}(B_\cap) = \mathcal{L}(B_1) \cap \mathcal{L}(B_2) \)
Intersection (General Case)

- Main problem: determining accepting states
  - need to go through accepting states of $B_1$ and $B_2$ infinite number of times
Intersection (General Case)

- Main problem: determining accepting states
  - need to go through accepting states of $B_1$ and $B_2$ infinite number of times

- Key idea: make 3 copies of the automaton:
  - 1st copy: start and accept here
  - 2nd copy: move from here from 1 when accepting state from $B_1$ has been seen
  - 3rd copy: move here from 2 when accepting state from $B_2$ has been seen, then go back to 1
Intersection (General Case)

Given two Büchi automata:

\[ B_1 = (\Sigma, Q_1, \Delta_1, Q^0_1, F_2) \quad B_2 = (\Sigma, Q_2, \Delta_2, Q^0_2, F_2) \]

Define

\[ B_\cap = (\Sigma, Q_1 \times Q_2 \times \{0, 1, 2\}, \Delta', Q^0_1 \times Q^0_2 \times \{0\}, Q_1 \times Q_2 \times \{2\}) \]

where

* \( (((s_1, s_2, 0), a, (s'_1, s'_2, 0)) \in \Delta' \text{ iff } (s_i, a, s'_i) \in \Delta_i, i = 1, 2, \text{ and } s'_1 \notin F_1 \)
* \( (((s_1, s_2, 1), a, (s'_1, s'_2, 1)) \in \Delta' \text{ iff } (s_i, a, s'_i) \in \Delta_i, i = 1, 2, \text{ and } s'_2 \notin F_2 \)
* \( (((s_1, s_2, 0), a, (s'_1, s'_2, 1)) \in \Delta' \text{ iff } (s_i, a, s'_i) \in \Delta_i, i = 1, 2, \text{ and } s'_1 \in F_1 \)
* \( (((s_1, s_2, 1), a, (s'_1, s'_2, 2)) \in \Delta' \text{ iff } (s_i, a, s'_i) \in \Delta_i, i = 1, 2, \text{ and } s'_2 \in F_2 \)
* \( (((s_1, s_2, 2), a, (s'_1, s'_2, 0)) \in \Delta' \text{ iff } (s_i, a, s'_i) \in \Delta_i, i = 1, 2 \)

Then, \( \mathcal{L}(B_\cap) = \mathcal{L}(B_1) \cap \mathcal{L}(B_2) \)
Complexity

- The emptiness problem for Büchi automata is decidable
  - $\mathcal{L}(A) \neq \emptyset$
  - logspace-complete for NLOGSPACE, i.e., solvable in linear time [Vardi, Wolper]) – see later in the lecture.

- Nonuniversality problem for Büchi automata is decidable
  - $\mathcal{L}(A) \neq \Sigma^\omega$
  - logspace-complete for PSPACE [Sistla, Vardi, Wolper]
Modeling Systems Using Automata

- A system is a set of all its executions. So, every state is accepting!

- Transform Kripke structure \((S, R, S_0, L)\), where \(L : S \rightarrow 2^{AP}\)

- ...into automaton \(A = (\Sigma, S \cup \{\ell\}, \Delta, \{\ell\}, S \cup \{\ell\})\),
  - where \(\Sigma = 2^{AP}\)
  - \((\ell, \alpha, s') \in \Delta \text{ iff } s \in S_0 \text{ and } \alpha = L(s)\)
  - \((s, \alpha, s) \in \Delta \text{ iff } (s, s') \in R \text{ and } \alpha = L(s')\)
LTL and Büchi Automata

- Specification – also in the form of an automaton!
- Büchi automata can encode all LTL properties.
- Examples:
  - $a U b$
  - Other examples:
    - $\square \lozenge p$
    - $\square \lozenge (p \lor q)$
    - $\neg \square \lozenge (p \lor q)$
    - $\neg(\square(p U q))$
Theorem [Wolper, Vardi, Sistla 83]: Given an LTL formula $\phi$, one can build a Büchi automaton $S = (\Sigma, Q, \Delta, Q_0, F)$, where

- $\Sigma = 2^{\text{Prop}}$
  - the number of automatic propositions, variables, etc. in $\phi$
- $|Q| \leq 2^{O(|\phi|)}$, where $|\phi|$ is the length of the formula

... s.t. $\mathcal{L}(S)$ is exactly the set of computations satisfying the formula $\phi$.

Algorithm: see Section 9.4 of Model Checking book or try one of the online tools:


But Büchi automata are more expressive than LTL!
Automata-theoretic Model Checking

- The system $A$ satisfies the specification $S$ when
  - $\mathcal{L}(A) \subseteq \mathcal{L}(S)$
  - ... each behavior of the system is among the allowed behaviours

- Alternatively,
  - let $\overline{\mathcal{L}(S)}$ be the language $\Sigma^\omega - \mathcal{L}(S)$. Then,
    - $\mathcal{L}(A) \subseteq \mathcal{L}(S) \iff \mathcal{L}(A) \cap \overline{\mathcal{L}(S)} = \emptyset$
    - no behavior of $A$ is prohibited by $S$
  - If the intersection is not empty, any behavior in it corresponds to a counterexample.
  - Counterexamples are always of the form $uv^\omega$, where $u$ and $v$ are finite words.
Complexity

- Checking whether a formula $\phi$ is satisfied by a finite-state model $K$ can be done in time $O(||K|| \times 2^{O(|\phi|)})$ or in space $O((log||K|| + ||\phi||)^2)$.

- i.e., checking is polynomial in the size of the model and exponential in the size of the specification.
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to a cycle containing an accepting state
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to a cycle containing an accepting state
- Is this automaton empty?
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to a cycle containing an accepting state
- Is this automaton empty?
  - No – it accepts $a(bef)^\omega$
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to a cycle containing an accepting state
- Is this automaton empty?
  - No – it accepts $a(bef)\omega$
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to a cycle containing an accepting state

- Is this automaton empty?
  - No – it accepts $a(bef)^\omega$
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to a cycle containing an accepting state
- Is this automaton empty?
  - No – it accepts $a(be f)^\omega$
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to a cycle containing an accepting state

- Is this automaton empty?
  - No – it accepts $a(bef)^\omega$
LTL Model-Checking

- LTL Model-Checking = Emptiness of Büchi automata
  - a tiny bit of automata theory +
  - trivial graph-theoretic problem
    - typical solution – use depth-first search (DFS)
- Problem: state-explosion
  - the graph is HUGE
- End result:
  - LTL model-checking a very elaborate DFS
Depth-First Search – Refresher
Depth-First Search – Refresher
Depth-First Search – Refresher
Depth-First Search – Refresher
Depth-First Search – Refresher
Depth-First Search – Refresher
Depth-First Search – Refresher

![Depth-First Search Graph]

1. Start at node 1
2. Visit node 2
3. Visit node 3
4. Visit node 4
5. Visit node 5
6. Visit node 6
Depth-First Search – Refresher

Diagram:

1 → 3 → 4 → 6 → 5
1 → 2
1 → 7 → 3

Depth-First Search – Refresher

Depth-first tree

1 <-> 2 <-> 3 -> 4 -> 5 -> 6

2 <-> 3

7

1 <-> 7
DFS – The Algorithm

1: \( \text{time} := 0 \)
2: proc \( \text{DFS}(v) \)
3: add \( v \) to \( \text{Visited} \)
4: \( d[v] := \text{time} \)
5: \( \text{time} := \text{time} + 1 \)
6: for all \( w \in \text{succ}(v) \) do
7: \( \text{if } w \notin \text{Visited} \text{ then} \)
8: \( \text{DFS}(w) \)
9: \( \text{end if} \)
10: end for
11: \( f[v] := \text{time} \)
12: \( \text{time} := \text{time} + 1 \)
13: end proc
DFS – Data Structures

- implicit STACK
  - stores the current path through the graph
- *Visited* table
  - stores visited nodes
  - used to avoid cycles
- for each node
  - *discovery time* – array $d$
  - *finishing time* – array $f$
What we want

- Running time
  - at most linear — anything else is not feasible

- Memory requirements
  - sequentially accessed – (for the STACK)
    - disk storage is good enough
    - assume unlimited supply – so can ignore
  - randomly accessed – (for hash tables)
    - must use RAM
    - limited resource – minimize
    - why cannot use virtual memory?
Additionally...

- Counterexamples
  - an automaton is non-empty iff exists an accepting run
  - this is the counterexample – we want it

- Approximate solutions
  - partial result is better than nothing!
DFS – Complexity

- Running time
  - each node is visited once
  - linear in the size of the graph

- Memory
  - the STACK
    - accessed sequentially
    - can store on disk – ignore
  - Visited table
    - randomly accessed – important
    - $|\text{Visited}| = S \times n$
    - $n$ – number of nodes in the graph
    - $S$ – number of bits needed to represent each node
Take 1 – Tarjan’s SCC algorithm

- Idea: find all maximal SCCs: $SCC_1$, $SCC_2$, etc.
  - an automaton is non-empty iff exists $SCC_i$ containing an accepting state
Take 1 – Tarjan’s SCC algorithm

- Idea: find all maximal SCCs: SCC₁, SCC₂, etc.
  - an automaton is non-empty iff exists SCCᵢ containing an accepting state
- Fact: each SCC is a sub-tree of DFS-tree
  - need to find roots of these sub-trees
Take 1 – Tarjan’s SCC algorithm

- Idea: find all maximal SCCs: SCC$_1$, SCC$_2$, etc.
  - an automaton is non-empty iff exists SCC$_i$ containing an accepting state
- Fact: each SCC is a sub-tree of DFS-tree
  - need to find roots of these sub-trees
Finding a Root of an SCC

• For each node $v$, compute $\text{lowlink}[v]$

• $\text{lowlink}[v]$ is the minimum of
  ◦ discovery time of $v$
  ◦ discovery time of $w$, where
    • $w$ belongs to the same SCC as $v$
    • the length of a path from $v$ to $w$ is at least 1

• Fact: $v$ is a root of an SCC iff
  ◦ $d[v] = \text{lowlink}[v]$
Finally: the algorithm

1: proc SCC_SEARCH(v)
2: add v to Visited
3: d[v] := time
4: time := time + 1
5: lowlink[v] := d[v]
6: push v on STACK
7: for all w ∈ succ(v) do
8: if w ∉ Visited then
9: SCC_SEARCH(w)
10: lowlink[v] := min(lowlink[v], lowlink[w])
11: else if d[w] < d[v] and w is on STACK then
12: lowlink[v] := min(d[w], lowlink[v])
13: end if
14: end for
15: if lowlink[v] = d[v] then
16: repeat
17: pop x from top of STACK
18: if x ∈ F then
19: terminate with “Yes”
20: end if
21: until x = v
22: end if
23: end proc
Finally: the algorithm

1: proc SCC_SEARCH(v)
2:    add v to Visited
3:    d[v] := time
4:    time := time + 1
5:    lowlink[v] := d[v]
6:    push v on STACK
7:    for all w ∈ succ(v) do
8:       if w ∉ Visited then
9:          SCC_SEARCH(w)
10:         lowlink[v] := min(lowlink[v], lowlink[w])
11: else if d[w] < d[v] and w is on STACK then
12:         lowlink[v] := min(d[w], lowlink[v])
13:    end if
14: end for
15: if lowlink[v] = d[v] then
16:    repeat
17:       pop x from top of STACK
18:       if x ∈ F then
19:           terminate with “Yes”
20:    end if
21: until x = v
22: end if
23: end proc
Tarjan’s SCC algorithm – Analysis

- **Running time**
  - linear in the size of the graph

- **Memory**
  - STACK – sequential, ignore
  - $Visited - O(S \times n)$
  - $lowlink - \log n \times n$
  - $n$ is not known a priori
    - assume $n$ is at least $\geq 2^{32}$

- **Counterexamples**
  - can be extracted from the STACK
  - even more – get multiple counterexamples

- If we sacrifice some of generality, can we do better?
Take 2 – Two Sweeps

- Don’t look for maximal SCCs
- Find a reachable accepting state that is on a cycle
- Idea: use two sweeps
  - sweep one: find all accepting states
  - sweep two: look for cycles from accepting states
Take 2 – Two Sweeps

- Don’t look for maximal SCCs
- Find a reachable accepting state that is on a cycle
- Idea: use two sweeps
  - sweep one: find all accepting states
  - sweep two: look for cycles from accepting states
- Problem?
  - no longer a linear algorithm (revisit the states multiple times)
Take 2 – Two Sweeps

- Don’t look for maximal SCCs
- Find a reachable accepting state that is on a cycle
- Idea: use two sweeps
  - sweep one: find all accepting states
  - sweep two: look for cycles from accepting states
- Problem?
  - no longer a linear algorithm (revisit the states multiple times)
Fixing non-linearity

• Graph Theoretic Result: let \( v \) and \( u \) be two nodes, such that
  - \( f[v] < f[u] \)
  - \( v \) is not on a cycle
  - then, no cycle containing \( u \) contains nodes reachable from \( v \)
Fixing non-linearity

• Graph Theoretic Result: let \( v \) and \( u \) be two nodes, such that
  - \( f[v] < f[u] \)
  - \( v \) is not on a cycle
  - then, no cycle containing \( u \) contains nodes reachable from \( v \)
Fixing non-linearity

• Graph Theoretic Result: let \( v \) and \( u \) be two nodes, such that
  - \( f[v] < f[u] \)
  - \( v \) is not on a cycle
  - then, no cycle containing \( u \) contains nodes reachable from \( v \)
Take 3 – Double DFS

1: proc DFS1(v)
2: add v to Visited
3: for all \( w \in \text{succ}(v) \) do
4: if \( w \notin \text{Visited} \) then
5: DFS1(w)
6: end if
7: end for
8: if \( v \in F \) then
9: add v to Q
10: end if
11: end proc

1: proc DFS2(v, f)
2: add v to Visited
3: for all \( w \in \text{succ}(v) \) do
4: if \( v = f \) then
5: terminate with “Yes”
6: else if \( w \notin \text{Visited} \) then
7: DFS2(w, f)
8: end if
9: end for
10: end proc

1: proc SWEEP2(Q)
2: while \( Q \neq \emptyset \) do
3: \( f := \text{dequeue}(Q) \)
4: DFS2(f, f)
5: end while
6: terminate with “No”
7: end proc

1: proc DDFS(v)
2: \( Q = \emptyset \)
3: \( \text{Visited} = \emptyset \)
4: DFS1(v)
5: \( \text{Visited} = \emptyset \)
6: SWEEP2(Q)
7: end proc
Double DFS – Analysis

• Running time
  ◦ linear! (single \textit{Visited} table for different final states, so no state is processed twice)

• Memory requirements
  ◦ $O(n \times S)$

• Problem
  ◦ where is the counterexample?!
Take 4 – Nested DFS

• Idea
  ◦ when an accepting state is finished
    • stop first sweep
  ◦ start second sweep
    • if cycle is found, we are done
  ◦ otherwise, restart the first sweep

• As good as double DFS, but
  ◦ does not need to always explore the full graph
  ◦ counterexample is readily available
    • a path to an accepting state is on the stack of the first sweep
    • a cycle is on the stack of the second sweep
A Few More Tweaks

- No need for two *Visited* hashtables
  - empty hashtable wastes space
  - merge into one by adding one more bit to each node
    - $(v, 0) \in V_isited$ iff $v$ was seen by the first sweep
    - $(v, 1) \in V_isited$ iff $v$ was seen by the second sweep

- Early termination condition
  - nested DFS can be terminated as soon as it finds a node that is on the stack of the first DFS
Nested DFS

1: proc $DFS1(v)$
2: add $(v, 0)$ to $Visited$
3: for all $w \in succ(v)$ do
4: if $(w, 0) \not\in Visited$ then
5: $DFS1(w)$
6: end if
7: end for
8: if $v \in F$ then
9: $DFS2(v, v)$
10: end if
11: end proc

1: proc $DFS2(v, f)$
2: add $(v, 1)$ to $Visited$
3: for all $w \in succ(v)$ do
4: if $v = f$ then
5: terminate with “Yes”
6: else if $(w, 1) \not\in Visited$ then
7: $DFS2(w, f)$
8: end if
9: end for
10: end proc
On-the-fly Model-Checking

• Typical problem consists of
  ◦ description of several processes $P_1, P_2, \ldots$
  ◦ property $\varphi$ in LTL

• Before applying DFS algorithm
  ◦ construct graph for $P = \prod_{i=1}^{n} P_i$
  ◦ construct Büchi automaton $A_{\neg \varphi}$ for $\neg \varphi$
  ◦ construct Büchi automaton for $P \cap A_{\neg \varphi}$
On-the-fly Model-Checking

- Typical problem consists of
  - description of several processes $P_1, P_2, \ldots$
  - property $\varphi$ in LTL
- Before applying DFS algorithm
  - construct graph for $P = \prod_{i=1}^{n} P_i$
  - construct Büchi automaton $A_{\neg \varphi}$ for $\neg \varphi$
  - construct Büchi automaton for $P \cap A_{\neg \varphi}$
- But,
  - all constructions can be done in DFS order
  - combine everything with the search
  - result: on-the-fly algorithm, only the necessary part of the graph is built
Symbolic LTL Model-Checking

- LTL Model-Checking = Finding a reachable cycle
- Represent the graph symbolically
  - and use symbolic techniques to search
- There exists an infinite path from $s$, iff $s \models EG \text{ true}$
  - the graph is finite
    - infinite $\Rightarrow$ cyclic!
  - exists a cycle containing an accepting state $a$ iff $a$ occurs infinitely often
    - use fairness to capture accepting states
- LTL Model-Checking = $EG \text{ true}$ under fairness!