Automata-Theoretic LTL Model Checking

Graph Algorithms for Software Model Checking

(based on Arie Gurfinkel's csc2108 project)

Emptiness of Büchi Automata

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

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

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 an accepting state,
- such that there exists a cycle containing it

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

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**

The result
- LTL model-checking is just a very elaborate DFS

Depth-First Search – Refresher
Depth-First Search – Refresher

Depth-First Search – Refresher
Depth-First Search – Refresher

1 -> 2
1 -> 3
3 -> 4
Depth-First Search – Refresher

Depth-First Search – Refresher
Depth-First Search – Refresher

depth-first tree
DFS – The Algorithm

1: proc DFS(v)
2: add v to Visited
3: d[v] := time
4: time := time + 1
5: for all w ∈ succ(v) do
6: if w ∉ Visited then
7: DFS(w)
8: end if
9: end for
10: f[v] := time
11: time := time + 1
12: 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 — like STACK
    - disk storage is good enough
    - assume unlimited supply — so can ignore
  - randomly accessed — like hash tables
    - must use RAM
    - limited resource — minimize
    - why cannot use virtual memory?

What else we want

- 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

[Diagram of a directed graph with labeled nodes and arrows showing connections.]

Automata-Theoretic LTL Model Checking – p.11
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

```plaintext
1: proc $\text{SCC\_SEARCH}(v)$
2:   add $v$ to $\text{Visited}$
3:   $d[v] := \text{time}$
4:   $\text{time} := \text{time} + 1$
5:   $\text{lowlink}[v] := d[v]$
6:   push $v$ on $\text{STACK}$
7:   for all $w \in \text{succ}(v)$ do
8:     if $w \not\in \text{Visited}$ then
9:       $\text{SCC\_SEARCH}(w)$
10:      $\text{lowlink}[v] := \min(\text{lowlink}[v], \text{lowlink}[w])$
11:    end if
12:   end for
13: end proc
```
Finally: the algorithm

1: proc SCC_SEARCH(v)
2: add v to Visited
3: \( d[v] := \text{time} \)
4: \( \text{time} := \text{time} + 1 \)
5: \( \text{lowlink}[v] := d[v] \)
6: push v on STACK
7: for all \( w \in \text{succ}(v) \) do
8: if \( w \notin \text{Visited} \) then
9: \( \text{SCC SEARCH}(w) \)
10: \( \text{lowlink}[v] := \min(\text{lowlink}[v], \text{lowlink}[w]) \)
11: else if \( d[w] < d[v] \) and \( w \) is on \( \text{STACK} \) then
12: \( \text{lowlink}[v] := \min(d[w], \text{lowlink}[v]) \)
13: end if
14: end for
15: if \( \text{lowlink}[v] = d[v] \) then
16: repeat
17: pop \( x \) from top of \( \text{STACK} \)
18: if \( x \in F \) 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
  - \( Visited - O(S \times n) \)
  - \( \text{lowlink} - \log n \times n \) (wasted space?)
  - \( n \) 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

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)

![Diagram of states](image.png)

Fixing non-linearity: Graph Theoretic Rest

- Fact: 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 $v$
Fact: 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: \hspace{1cm} add $v$ to $Visited$
3: \hspace{1cm} for all $w \in \text{succ}(v)$ do
4: \hspace{2cm} if $w \not\in Visted$ then
5: \hspace{3cm} $DFS1(w)$
6: \hspace{2cm} end if
7: \hspace{1cm} end for
8: \hspace{1cm} if $v \in F$ then
9: \hspace{2cm} add $v$ to $Q$
10: \hspace{1cm} end if
11: end proc

1: proc $DFS2(v, f)$
2: \hspace{1cm} add $v$ to $Visited$
3: \hspace{1cm} for all $w \in \text{succ}(v)$ do
4: \hspace{2cm} if $v = f$ then
5: \hspace{3cm} terminate with “Yes”
6: \hspace{2cm} else if $w \not\in Visted$ then
7: \hspace{3cm} $DFS2(w, f)$
8: \hspace{2cm} end if
9: \hspace{1cm} end for
10: end proc

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

Double DFS – Analysis

- Running time
  - linear! (single $Visited$ 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

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 \text{Visited} \text{ iff } v \text{ was seen by the first sweep}
  - \((v, 1) \in \text{Visited} \text{ iff } v \text{ 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
On-the-fly Model-Checking

Typical problem consists of
- description of several process $P_1, P_2, \ldots$
- 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
State Explosion Problem

- the size of the graph to explore is huge
- on real programs
  - DFS dies after examining just 1% of the state space
- What can be done?
  - abstraction
    - false negatives
  - partial order reduction. (to be covered)
    - exact – but not applicable to full LTL
  - partial exploration – explore as much as possible
    - false positives
- In practice – combine all 3

Partial exploration techniques

- Explore as much of the graph as possible
- The requirements
  - must be compatible with
    - on-the-fly model-checking
    - nested depth-first search
  - size of the graph not known a priori
    - must perform as good as full exploration when enough memory is available
    - must degrade gracefully
- We will look at two techniques
  - bitstate hashing
  - hashcompact – a type of state compression
Bitstate Hashing

- A hashtable is
  - An array $d$ of $k$ entries
  - A hash function $\text{hash} : \text{States} \rightarrow 0..k - 1$
  - A collision resolution protocol

To insert $v$ into a hashtable
- Compute $\text{hash}(v)$
- If $d[\text{hash}(v)]$ is empty, $d[\text{hash}(v)] = v$
- Otherwise, apply collision resolution

To lookup $v$
- If $d[\text{hash}(v)]$ is empty, $v$ is not in the table
- Else if $d[\text{hash}(v)] = v$, $v$ is in the table
- Otherwise, apply collision resolution

If there are no collisions, don’t need to store $v$ at all!
- Instead, just store one bit – empty or not
- Even better, use two hash functions
  - To insert $v$, set $d[\text{hash}_1(v)] = 1$ and $d[\text{hash}_2(v)] = 1$
- Sound with respect to false answers
  - If a counterexample is found, it is found!
- In practice, up to 99% coverage
- Collisions increase gradually when not enough memory
- Coverage decreases at the rate collisions increase
Why does this work?

- If nested DFS stops when a successor to \( v \) in \( DFS2 \) is on the stack of \( DFS1 \), how is soundness guaranteed, i.e., why is the counterexample returned by model-checker real?
- Answer: States are stored on the stack without hashing, since stack space does not need to be saved.

Hashcompact

- Assume a large virtual hashtable, say \( 2^{64} \) entries
- For each node \( v \),
  - instead of using \( v \),
  - use \( hash(v) \), its hash value in the large table
- Store \( hash(v) \) in a normal hashtable,
  - or even the one with bitstate hashing
- When there is enough memory
  - probability of missing a node is \( < 10^{-3} \)
- Degradation
  - expected coverage decreases rapidly, when not enough memory
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 \( ||EG \text{ true}||(s) \)
  - 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!

food for slide eater