Automata-Theoretic LTL Model Checking

Graph Algorithms for Software Model Checking

(based on Arie Gurfinke'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 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: $\text{SCC}_1$, $\text{SCC}_2$, etc.
  - an automaton is non-empty iff exists $\text{SCC}_i$
    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
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 $\text{SCC\_SEARCH}(v)$
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 $w \notin \text{Visited}$ then
9:     $\text{SCC\_SEARCH}(w)$
10:    end if
11:   end for
12:  $\text{lowlink}[v] := \min(\text{lowlink}[v], \text{lowlink}[w])$
13:  end if
14: end for
15: end if
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
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 – $\log n \times n$ (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)

![Graph Diagram]

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 ∈ succ(v) do
4:     if w ∉ Visited then
5:       DFS1(w)
6:   end if
7:   end for
8:   if v ∈ F then
9:     add v to Q
10: end if
11: end proc

1: proc SWEEP2(Q)
2:   while Q ≠ [] do
3:     f := dequeue(Q)
4:     DFS2(f, f)
5:   end while
6:   terminate with “No”
7: end proc

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

1: proc DDFS(v)
2:   Q = ∅
3:   Visited = ∅
4:   DFS1(v)
5:   Visited = ∅
6:   SWEEP2(Q)
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 \textit{Visited} hashtables
  - empty hashtable wastes space
  - merge into one by adding one more bit to each node
    - \((v, 0) \in \text{Visited} \iff v \text{ was seen by the first sweep}
    - \((v, 1) \in \text{Visited} \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 $P = \Pi_{i=1}^nP_i$
  - 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 $hash : States \rightarrow 0..k - 1$
  - a collision resolution protocol
- to insert $v$ into a hashtable
  - compute $hash(v)$
  - if $d[hash(v)]$ is empty, $d[hash(v)] = v$
  - otherwise, apply collision resolution
- to lookup $v$
  - if $d[hash(v)]$ is empty, $v$ is not in the table
  - else if $d[hash(v)] = v$, $v$ 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 $d[hash_1(v)] = 1$ and $d[hash_2(v)] = 1$
- 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 \( DFS_2 \) is on the stack of \( DFS_1 \), 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 $||EG$ 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 = $EG$ true under fairness!

food for slide eater