Explicit State Reachability Analysis
Overview

- How to analyze properties of a model?
  - Today: Explicit state model checking
  - Reachability Analysis
  - (In between, certain graph traversal algorithms)
Model Checking

- So far, we have seen ways to specify the properties.
- Model checking addresses (efficient) algorithms to decide whether the system satisfies the property.
- First approach: Explicit state model checking
Reachability in Finite-State Machines

- Reachability analysis
  - Central in model checking
  - Safety properties
- “Can a set of states ever be reached starting from the initial states?”
  - Usually interested in reachability of unsafe or “bad” states.
Algorithm: Reachability in FSMs

\[
\text{Reach} = S_0
\]

// Assuming \( S_0 \cap \text{Bad} = \emptyset \)

Queue = \( S_0 \)

while Queue \( \neq \emptyset \) do

remove \( s \) from Queue

for all \((s, t) \in R\) do

if \( t \in \text{Bad} \) then

exit(“System is unsafe”)

end if

if \( t \notin \text{Reach} \) then

add \( t \) to \text{Reach}

add \( t \) to Queue

end if

end for

end while

exit(“System is safe”)

Red: Bad states
Yellow: Reached states
Algorithm: Reachability in FSMs

\[
\begin{align*}
\text{Reach} &= S_0 \\
\text{Queue} &= S_0 \\
\text{while } &\text{Queue }\neq \emptyset \text{ do} \\
&\text{remove } s \text{ from Queue} \\
&\text{for all } (s, t) \in R \text{ do} \\
&\quad \text{if } t \in \text{Bad} \text{ then} \\
&\quad \quad \text{exit(“System is unsafe”)} \\
&\quad \text{end if} \\
&\quad \text{if } t \not\in \text{Reach} \text{ then} \\
&\quad \quad \text{add } t \text{ to Reach} \\
&\quad \quad \text{add } t \text{ to Queue} \\
&\quad \text{end if} \\
&\text{end for} \\
&\text{end while} \\
&\text{exit(“System is safe”)}
\end{align*}
\]
Algorithm: Reachability in FSMs

Reach = S₀
Queue = S₀
while Queue ≠ ∅ do
    remove s from Queue
    for all (s, t) ∈ R do
        if t ∈ Bad then
            exit(“System is unsafe”)  
        end if
        if t ∉ Reach then
            add t to Reach
            add t to Queue
        end if
    end for
end while
exit(“System is safe”)
Algorithm: Reachability in FSMs

Reach = S₀
Queue = S₀
while Queue ≠ ∅ do
    remove s from Queue
    for all (s, t) ∈ R do
        if t ∈ Bad then
            exit("System is unsafe")
        end if
        if t ∉ Reach then
            add t to Reach
            add t to Queue
        end if
    end for
end while
exit("System is safe")

Queue = {s₁}

Red: Bad states
Yellow: Reached states
Algorithm: Reachability in FSMs

```
Reach = S_0
Queue = S_0
while Queue ≠ ∅ do
    remove s from Queue
    for all (s, t) ∈ R do
        if t ∈ Bad then
            exit("System is unsafe")
        end if
        if t ∉ Reach then
            add t to Reach
            add t to Queue
        end if
    end for
end while
exit("System is safe")
```

Red: Bad states
Yellow: Reached states
Algorithm: Reachability in FSMs

```
Reach = S₀
Queue = S₀
while Queue ≠ ∅ do
    remove s from Queue
    for all (s, t) ∈ R do
        if t ∈ Bad then
            exit(“System is unsafe”)
        end if
        if t ∉ Reach then
            add t to Reach
            add t to Queue
        end if
    end for
end while
exit(“System is safe”)
```
Algorithm: Reachability in FSMs

```
Reach = S_0
Queue = S_0
while Queue ≠ ∅ do
    remove s from Queue
    for all (s, t) ∈ R do
        if t ∈ Bad then
            exit(“System is unsafe”)
        end if
        if t ∉ Reach then
            add t to Reach
            add t to Queue
        end if
    end for
end while
exit(“System is safe”)
```

Queue = ∅

Red: Bad states
Yellow: Reached states
Algorithm: Reachability in FSMs

Reach = S₀
Queue = S₀
while Queue ≠ ∅ do
    remove s from Queue
    for all (s, t) ∈ R do
        if t ∈ Bad then
            exit(“System is unsafe”)  
        end if
        if t ∉ Reach then
            add t to Reach
            add t to Queue
        end if
    end for
end while
exit(“System is safe”)
Forward Reachability

- Properties can be analyzed by enumerating all reachable states from the initial state.
- For safety properties, it is enough to check each reached state for its safety.
Backward Reachability

- Properties can also be analyzed by enumerating all states that can reach the unsafe state
  - Iteratively compute the predecessors of states that can end up in an unsafe state
  - For safety, check that none of these states is an initial state
  - Sometimes, could be faster than forward reachability (less states to enumerate)
Review: Depth-First Search vs. Breadth-First Search

- **Graph Traversal algorithms**
  - **Depth-First Search (DFS)**
    - Use stack (LIFO) to keep states that need further processing
    - LIFO: Last-in first-out
    - Searches state space by following single path at a time
  - **Breadth-First Search (BFS)**
    - Use FIFO-queue to keep states that need further processing
    - FIFO: First-in first-out
    - Searches state space by processing rounds of forward steps
Review: Potential DFS Traversal

- A (Start at some initial state)
- A → B (pick a child)
- A → B → D (pick a child)
- A → B (after backtrack)
- A → B → E (pick other child)
- A → B (after backtrack)
- A (after backtrack)
- A → C (pick other child)
- A → C → F (pick a child)
- A → C (after backtrack)
- A (after backtrack)
- Done (after backtrack)
Review: Potential BFS Traversal

- A (Start at some initial state)
- Put all unseen children in queue: B, C
- Remove B from queue; add all its unseen children into queue: C, D, E
- Remove C; add all its unseen children into queue: D, E, F
- Remove D; add all its unseen children into queue: E, F
- Remove E; add all its unseen children into queue: F
- Remove F; all all its unseen children into queue: Ø
- Done
Counterexamples

- When properties are violated, model checker can provide a bug trace from the state exploration.
- Very useful to analyze the problematic behavior:
  - Could be actual problem in system
  - Sometimes due to incorrect environment modeling
- Counterexample is used to localize the fault and fix it.
- Generally speaking, counterexamples are the most useful feature of model checkers for actual designers/users:
  - Many counterexamples discovered early in design stage
  - Often more useful than final “proved” answer
Computing Counterexamples

- Reachability algorithm implementations: DFS or BFS
- **DFS – Depth-First Search**
  - Stack content represents counterexample from initial state to unsafe state
  - No additional information needs to be stored to compute counterexample
  - Memory efficient
- **BFS – Breadth-First Search**
  - Finds a shortest counterexample
  - Requires additional information to compute counterexample
  - Needs to store for each reachable state which its predecessor is
Computing Counterexamples using DFS

```
Reach = ∅  // Assuming S₀ ∩ Bad = ∅
Create an empty stack
while Reach ⊈ S₀ do
    pick s₀ ∈ S₀ \ Reach
    put s₀ onto stack
    while stack is not empty do
        if ∀(top(stack), t) ∈ R : t ∈ Reach then
            pop stack
        else
            pick t ⊈ Reach, s.t. (top(stack), t) ∈ R
            add t to Reach
            push t onto stack
        if t ∈ Bad then
            print stack
            exit("System is unsafe")
        end if
    end while
end while
exit("System is safe")
```

Red: Bad states
Yellow: Reached states
Computing Counterexamples using DFS

Reach = ∅
Create an empty stack

while Reach ∉ S₀ do
    pick s₀ ∈ S₀ \ Reach
    put s₀ onto stack
    while stack is not empty do
        if ∀(top(stack), t) ∈ R : t ∈ Reach then
            pop stack
        else
            pick t ∉ Reach, s.t. (top(stack), t) ∈ R
            add t to Reach
            push t onto stack
            if t ∈ Bad then
                print stack
                exit(“System is unsafe”)
            end if
        end if
    end while
end while
exit(“System is safe”)
Computing Counterexamples using DFS

Reach = ∅
Create an empty stack
while Reach ⋈ S₀ do
  pick s₀ ∈ S₀ \ Reach
  put s₀ onto stack
  while stack is not empty do
    if ∀(top(stack), t) ∈ R : t ∈ Reach then
      pop stack
    else
      pick t ⋈ Reach, s.t. (top(stack), t) ∈ R
      add t to Reach
      push t onto stack
      if t ∈ Bad then
        print stack
        exit(“System is unsafe”)
      end if
    end if
  end while
end while
exit(“System is safe”)
Computing Counterexamples using DFS

Reach = ∅
Create an empty stack
while Reach ∉ S₀ do
    pick s₀ ∈ S₀ \ Reach
    put s₀ onto stack
    while stack is not empty do
        if ∀(top(stack), t) ∈ R : t ∈ Reach then
            pop stack
        else
            pick t ∉ Reach, s.t. (top(stack), t) ∈ R
            add t to Reach
            push t onto stack
        if t ∈ Bad then
            print stack
            exit("System is unsafe")
    end if
end if
end while
end while
exit("System is safe")
Computing Counterexamples using DFS

Reach = ∅
Create an empty stack
while Reach ⊈ S₀ do
    pick s₀ ∈ S₀ \ Reach
    put s₀ onto stack
    while stack is not empty do
        if ∀(top(stack), t) ∈ R : t ∈ Reach then
            pop stack
        else
            pick t ⊈ Reach, s.t. (top(stack), t) ∈ R
            add t to Reach
            push t onto stack
            if t ∈ Bad then
                print stack
                exit("System is unsafe")
            end if
        end if
    end while
end while
exit("System is safe")
Computing Counterexamples using DFS

Reach = ∅
Create an empty stack

while Reach ⊈ S₀ do
    pick s₀ ∈ S₀ \ Reach
    put s₀ onto stack
    while stack is not empty do
        if ∀(top(stack), t) ∈ R : t ∈ Reach then
            pop stack
        else
            pick t ∈ Reach, s.t. (top(stack), t) ∈ R
            add t to Reach
            push t onto stack
            if t ∈ Bad then
                print stack
                exit("System is unsafe")
            end if
        end if
    end while
end while
exit("System is safe")
Case Study: NASA’s PathFinder on Mars, 1997
Case Study: NASA’s PathFinder on Mars, 1997

- Design fault caused rover to loose contact with earth
  - Priority inversion problem caused intermittent deadlock
  - Global real-time operating system would discover malfunction and initiate system reset
  - Bug was traced through logging mechanism and fixed with software update from earth

- Problem description presented here highly simplified
  - Based on Holzmann: “The Spin Model Checker”, chapter 5.
Modeling NASA’s Pathfinder Problem

High priority process producing data in Running state:

- Idle ➔ Waiting
  - mutex := free
- Waiting ➔ Running
  - mutex := busy

Communication between processes occurs through global, shared variable mutex, which is initially set to free.

Low priority process consuming data in Running state:

- Idle ➔ Waiting
  - highState := Idle
- Waiting ➔ Running
  - highState := Idle
  - mutex := free
- Running ➔ Waiting
  - highState := Idle
  - mutex := free ➔ mutex := busy

Low priority process is blocked from execution by priority-driven real-time operating system.

Each action can only occur, when high priority process is not producing new data, that is when it is in its Idle state.
System Model

- Typically, an overall system is specified as a collection of modules and the environment of the system.
- Each module is modeled as an automaton.
- There are two ways of constructing overall system model:
  - Synchronous composition
  - Asynchronous composition
Deadlock
A Property To Check For: Deadlock
Synchronous Product

- Often used in modeling hardware
- At each step, all modules proceed in lock-step
- Given two structures $M_i = (S_i, S_{i0}, R_i)$, the synchronous product is defined as $M = (S, S_0, R)$ using

$$S = S_1 \times S_2$$
$$S_0 = S_{10} \times S_{20}$$
$$R((s_1, s_2), (t_1, t_2)) \iff R_1(s_1, t_1) \land R_2(s_2, t_2)$$
Synchronous Model Example

\begin{itemize}
  \item $M_S = (S, S_0, R_S)$
  \item $S = \{s_0, s_1, s_2\}$
  \item $S_0 = \{s_0\}$
  \item $M_T = (T, T_0, R_T)$
  \item $T = \{t_0, t_1\}$
  \item $T_0 = \{t_0\}$
\end{itemize}

Synchronous product of $M_S$ and $M_T$:

$M = (Q, Q_0, R)$

$Q = \{(s_0, t_0), (s_0, t_1), (s_1, t_0),$
$\quad (s_1, t_1), (s_2, t_0), (s_2, t_1)\}$

$Q_0 = \{(s_0, t_0)\}$

$R$ as shown.
Asynchronous Product

- Often used to model software (interleaved model)
- At each time step, one module is chosen randomly, which can proceed a single step
- Given two structures $M_i=(S_i, S_{i0}, R_i)$, the asynchronous product is defined as $M=(S, S_0, R)$ using

$$S = S_1 \times S_2$$
$$S_0 = S_{10} \times S_{20}$$
$$R((s_1, s_2), (t_1, t_2)) \text{ iff } [ R_1(s_1, t_1) \land s_2 = t_2 ] \lor [ R_2(s_2, t_2) \land s_1 = t_1 ]$$
Asynchronous Model Example

Asynchronous product of $M_S$ and $M_T$: 
$M=(Q,Q_0,R)$ 
$Q=\{(s_0,t_0),(s_0,t_1),(s_1,t_0),(s_1,t_1),(s_2,t_0),(s_2,t_1)\}$ 
$Q_0=\{(s_0,t_0)\}$ 
$R$ as shown.
Asynchronous Example: Software

Process 1:
```
int x = 0;
extern int y;
(1) x = 1;
(2) y = 2;
(3) x++;
(4) y++;
```

Process 2:
```
int y = 0;
extern int x;
(5) y = 3;
(6) x = 2;
(7) y++;
(8) x++;
```

pc1, pc2, x, y

1,5,0,0

2,5,1,0

3,5,1,2

4,5,2,2

F,5,2,3

5,2,3

1,5,2,3

3,6,1,2

2,7,1,4

2,7,1,3

1,7,2,3

1,8,2,4

3,7,2,4

2,8,2,4

2,8,1,4

1,F,3,4

1,5,2,3
A state is deadlocked, when there is no outgoing transition possible.
Here: Deadlock reached in state when high priority process is Waiting, while low priority process was running, but cannot complete due to high priority lock.