Logic Model Checking

Lecture Notes 15:18

Caltech 101b.2

January-March 2005

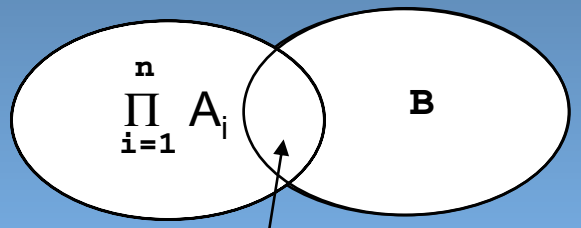
Course Text:

The Spin Model Checker: Primer and Reference Manual

Addison-Wesley 2003, ISBN 0-321-22862-6, 608 pgs.

automata products

- consider a system of n processes, modeled as finite state automata A_1, A_2, ... A_n
- add property automaton B (e.g. derived from an LTL formula)
- a model checker can compute the reachable state space as
  - $S = B \otimes \prod_{i=1}^{n} A_i$



an *asynchronous* product of n automata

a *synchronous* product of 2 automata

S: another finite state automaton: an ω-automaton representing the *relevant portion* of the global statespace i.e., as defined by B

asynchronous product

- the *asynchronous* product Π of a finite set of finite automata A_1, A_2, ... A_n is a new finite state automaton

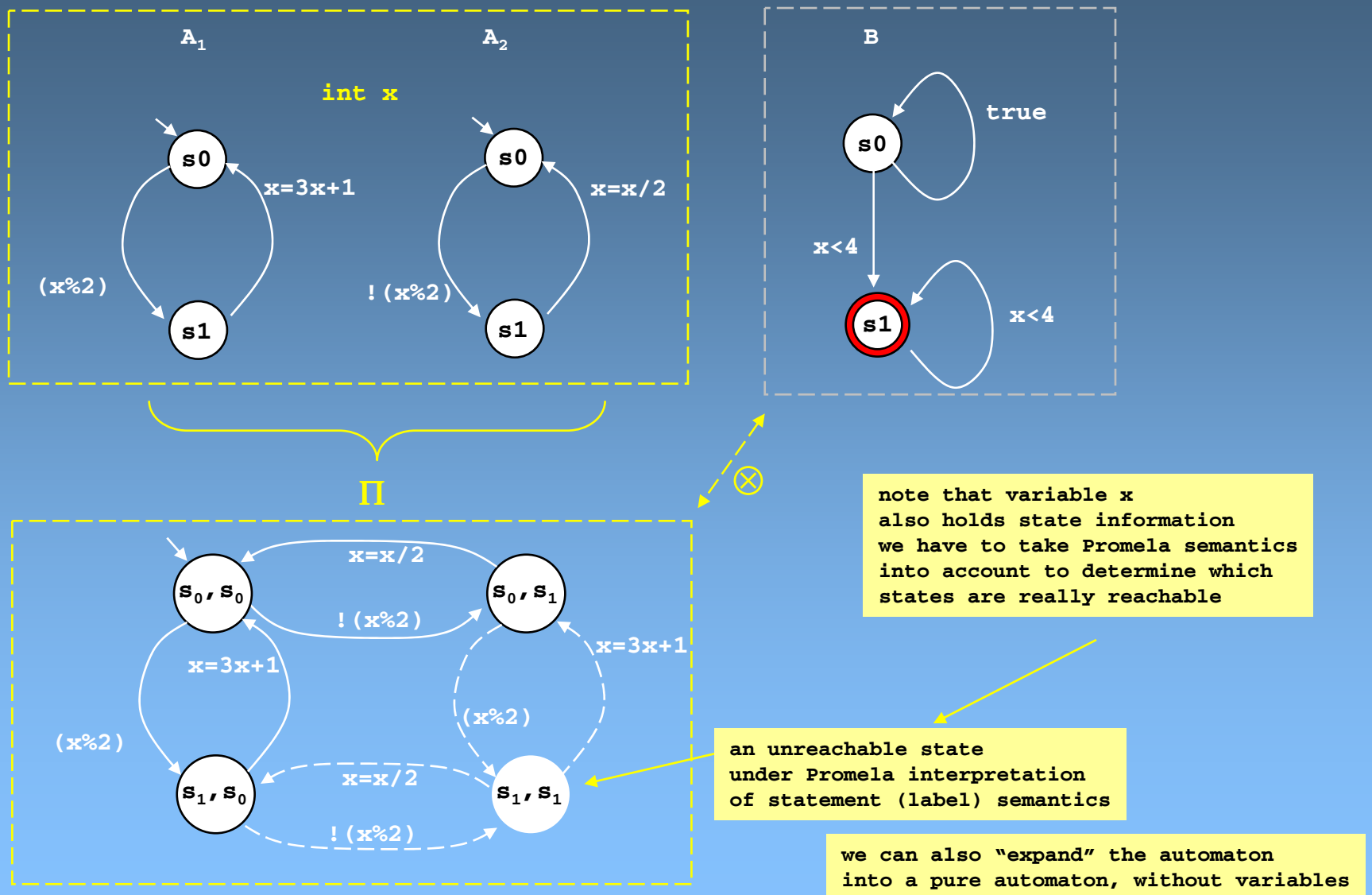
  $A = \{ S, s_0, L, T, F)$ where

$A.S$	is the Cartesian product $A_1.S \times A_2.S \times ... \times A_n.S$
$A.s_0$	is the n-tuple $(A_1.s_0, A_2.s_0, ... , A_n.s_0)$
$A.L$	is the union set $A_1.L \cup A_2.L \cup ... \cup A_n.L$
$A.T$	is the set of tuples $((x_1,...,x_n),l,(y_1,...y_n))$ such that $\exists i, 1 \le i \le n, (x,l,y) \in A_i.T$ and $\forall j, 1 \le j \le n, j \ne i \rightarrow (x_j \equiv y_j)$
$A.F$	contains those states from A.S that satisfy $\forall(A_1.s, A_2.s, ... , A_n.s) \in A.F \rightarrow \exists i, 1 \le i \le n, A_i.s \in A_i.F$

note that *not* all states in A.S or A.F are necessarily reachable from $A.s_0$

atomic and rv handshakes are defined through the executability rules: pre-condition and effect clauses

small example



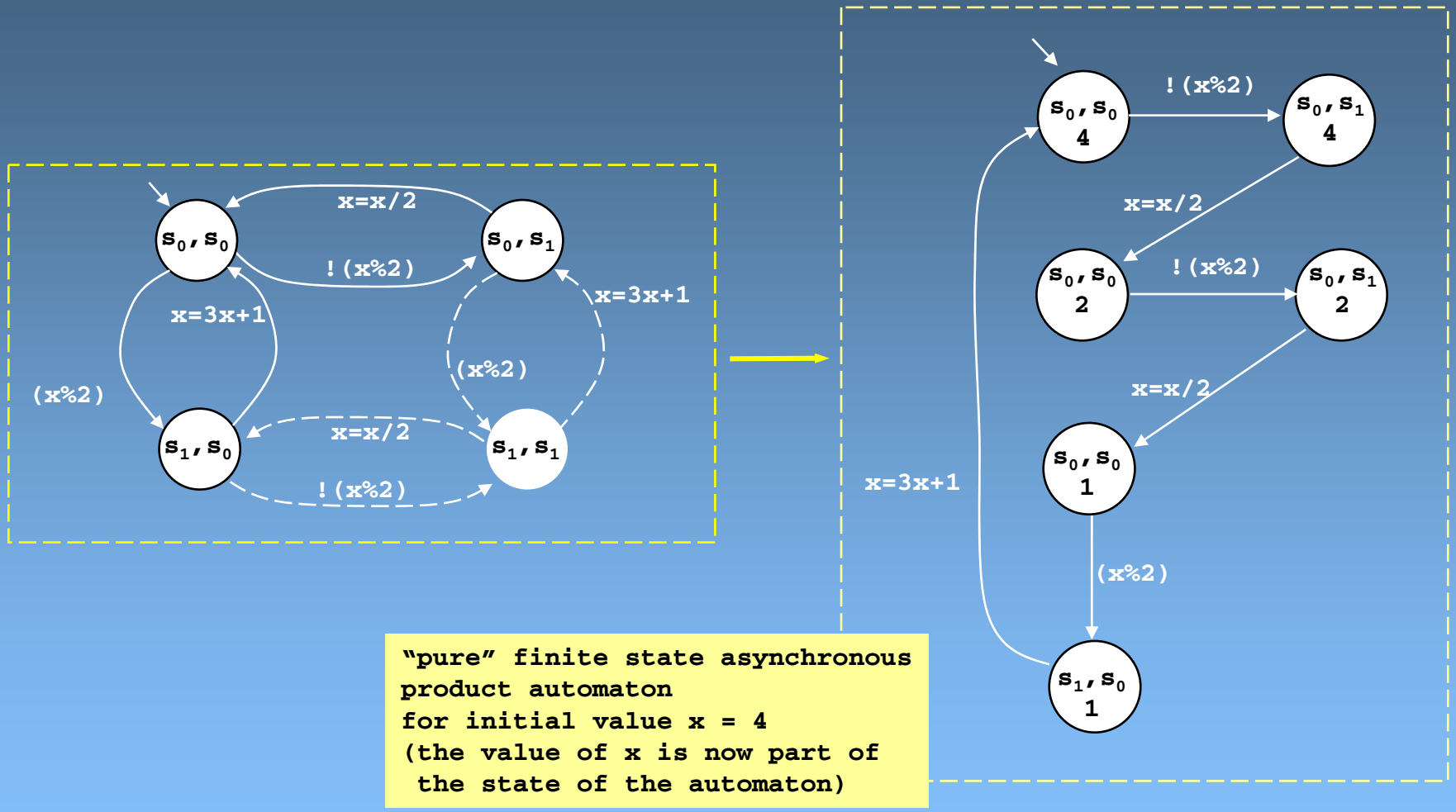
note that variable x
also holds state information
we have to take Promela semantics
into account to determine which
states are really reachable

an unreachable state
under Promela interpretation
of statement (label) semantics

we can also "expand" the automaton
into a pure automaton, without variables

small example

book p. 558-559



"pure" finite state asynchronous product automaton for initial value x = 4 (the value of x is now part of the state of the automaton)

synchronous product
(the never claim observer)

- the *synchronous* product of a finite set of finite automata P and B is the finite state automaton

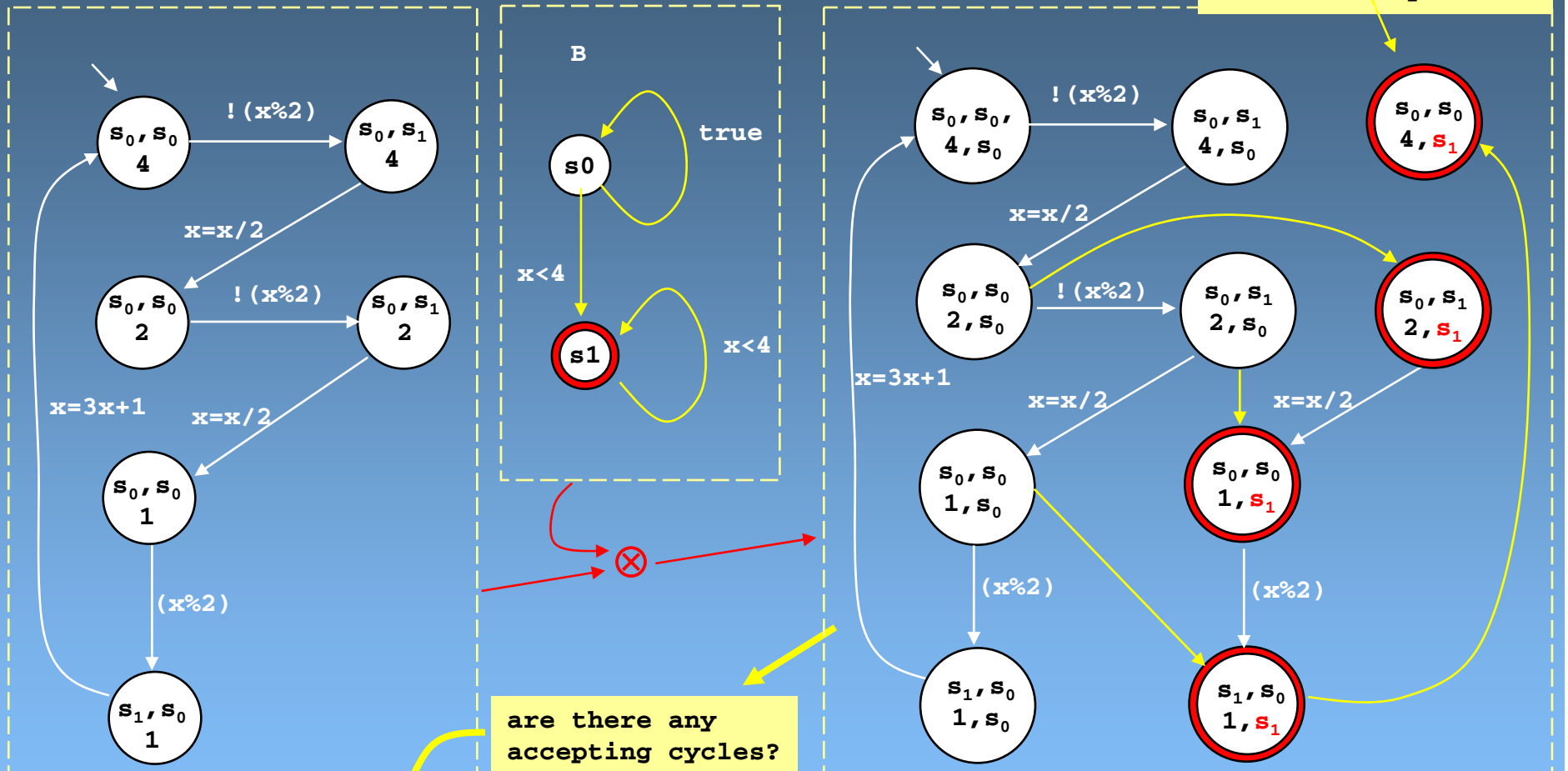
  A = { S, s_0, L, T, F) where

A.S	is the Cartesian product P'.S x B.S where P' is the *stutter closure* of P (a nil self-loop is attached to every state in P *without* outgoing transitions in P.T)
A.s_0	is the pair (P.s_0,B.s_0)
A.L	is the set of pairs (l_1,l_2) such that $l_1 \in$ P'.L and $l_2 \in$ B.L
A.T	is the set of pairs (t_1,t_2) such that $t_1 \in$ P'.T and $t_2 \in$ B.T
A.F	is the set of pairs (s_1,s_2) such that $s_1 \in$ P'.F *or* $s_2 \in$ B.F

this is the basic automaton structure.
we can next *interpret* the labels under Promela semantics to eliminate all transitions from A.T where either l_1 or l_2 is not executable
(l_2 is usually a boolean condition)

not all states in A.S or A.F are necessarily reachable from A.s_0 under the chosen interpretation of labels

the example: $B \otimes \prod_{i=1}^{2} A_i$



search algorithms

- checking safety properties
  - basic depth-first search
  - variant1: stateless search
  - variant2: depth-limited search
  - breadth-first search
- checking liveness properties
  - non-progress cycles
  - acceptance cycles
  - Spin's nested depth-first search algorithm
- adding fairness constraints
  - Choueka's flag construction method
- optimization methods
  - partial order reduction, state compression, alternate state representation methods

the search problem

- given a product automaton ($B \otimes \Pi A_i$)
  - find all runs that violate a safety property
    - 'bad' reachable states
- given a product automaton ($B \otimes \Pi A_i$)
  - find all runs that violate a liveness property
    - accepting ω-runs

- some observations:
  - these are basic graph search problems
    - but the graphs to be searched can be *very* large
  - we want to avoid having to construct the graph first
    - and if we need to store anything about the graph, it has to be as little information as possible (so that we can handle larger problems)

basic depth-first search

```
Automaton A = { S, s0, L, T, F }
Stack D = {}
Statespace V = {}

Start()
{
    Add_Statespace(V, A.s0 )
    Push_Stack(D, A.s0 )
    Search()
}

Search()
{
    s = Top_Stack(D)
    for each (s,l,s')∈ A.T
        if In_Statespace(V, s')== false
        {   Add_Statespace(V, s')
            Push_Stack(D, s')
            Search()
        }
    Pop_Stack(D)
}
```

```
Push_Stack(D,s)
            adds s to ordered set D
In_Stack(D,s)
            true iff s is in D
Top_Stack(D,s)
            returns top element in D
            if any
Pop_Stack(D)
            removes top element from D
            if any
```

```
Add_Statespace(V,s)
            adds s to set V
In_Statespace(V,s)
            true iff s is in V
```

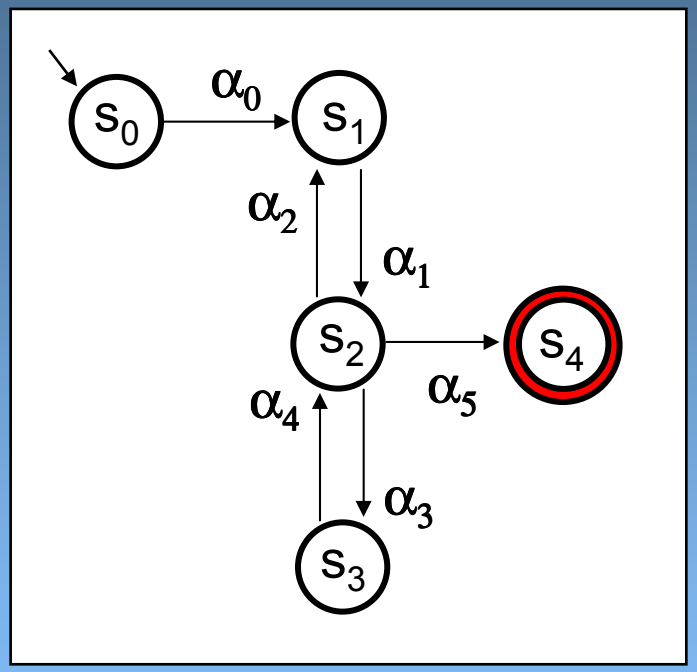
the DFS is most easily written as a recursive procedure -- but the actual Spin implementation is iterative (originally to increase efficiency)

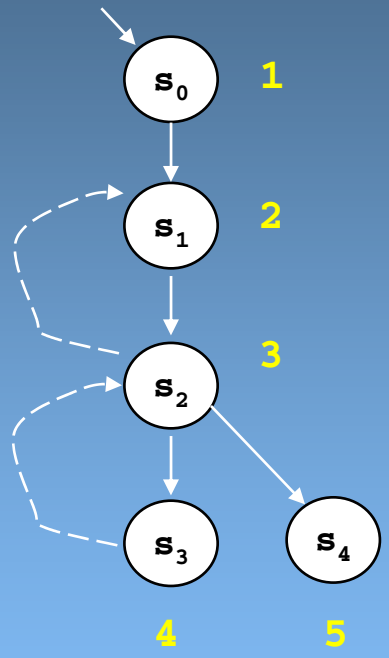
Fig. 8.1 p. 168

objective:
-store as little data about the graph as possible
- stores *states* in V, but not *transitions*
- Statespace V is there to prevent doing redundant work
- for correctness, V does not need to be complete
- in fact, V does not need to be there at all....

depth-first search order

depth-first search numbers





checking safety properties

(properties of states)

```
Automaton A = { S, s0, L, T, F }
Stack D = {}
Statespace V = {}

Start()
{
    Add_Statespace(V, A.s0 )
    Push_Stack(D, A.s0 )
    Search()
}

Search()
{
    s = Top_Stack(D)

    if (!Safety(s))
        Print_Stack(D)

    for each (s,l,s')∈ A.T
        if In_Statespace(V, s')== false
        {   Add_Statespace(V, s')
            Push_Stack(D, s')
            Search()
        }
    Pop_Stack(D)
}
```

Fig. 8.2, p. 170

assertion violations
invalid endstates
termination of a never claim

prints out the elements of stack D, from bottom to top, giving the complete counter-example / error scenario for the safety violation

a stateless search

(memory efficient, but *excessively* time consuming...)

no Statespace V

```
Automaton A = { S, s0, L, T, F }
Stack D = {}
/* Statespace V = {} */

Start()
{
    Push_Stack(D, A.s0 )
    Search()
}

Search()
{
    s = Top_Stack(D)
    for each (s,l,s')∈ A.T
        if In_Stack(D, s')== false
        {   Push_Stack(D, s')
            Search()
        }
    Pop_Stack(D)
}
```

replaced In_Statespace(V,s') with In_Stack(D,s')

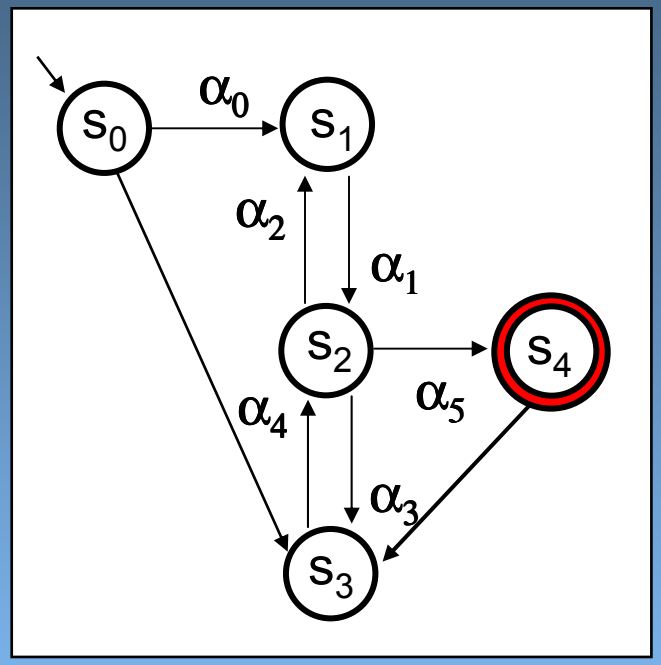
Fig. 8.5 p. 176

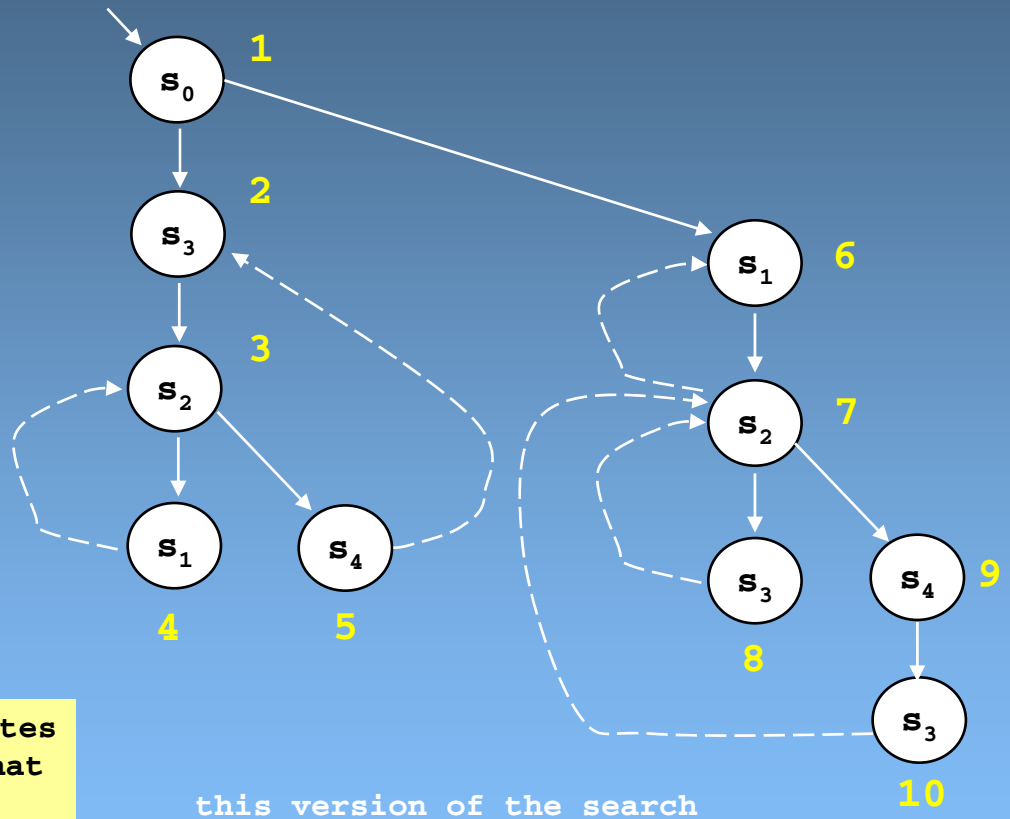
the algorithm is still guaranteed to terminate in a finite number of steps

Statespace V is used to prevent doing redundant work
- for correctness, it does not need to be complete
- in fact, it does not need to be there at all....

depth-first search order

extra work when *not* using the statespace construct





if s_3 has a sub-tree of 100,000 states the stateless search would visit that entire subtree at least 3 times...

-> maintaining the statespace cache is an optimization technique - the precise method in which the cache is maintained gives us lots of choices

this version of the search visits 10 instead of 5 states... (doing redundant work)
s_3 is visited 3 times here

checking safety properties

with a stateless search

```
Automaton A = { S, s0, L, T, F }
Stack D = {}
/* StateSpace V = {} */

Start()
{
    Push_Stack(D, A.s0 )
    Search()
}

Search()
{
    s = Top_Stack(D)

    if (!Safety(s))
          Print_Stack(D)

    for each (s,l,s')∈ A.T
          if In_Stack(D, s')== false
          {    Push_Stack(D, s')
               Search()
          }
    Pop_Stack(D)
}
```

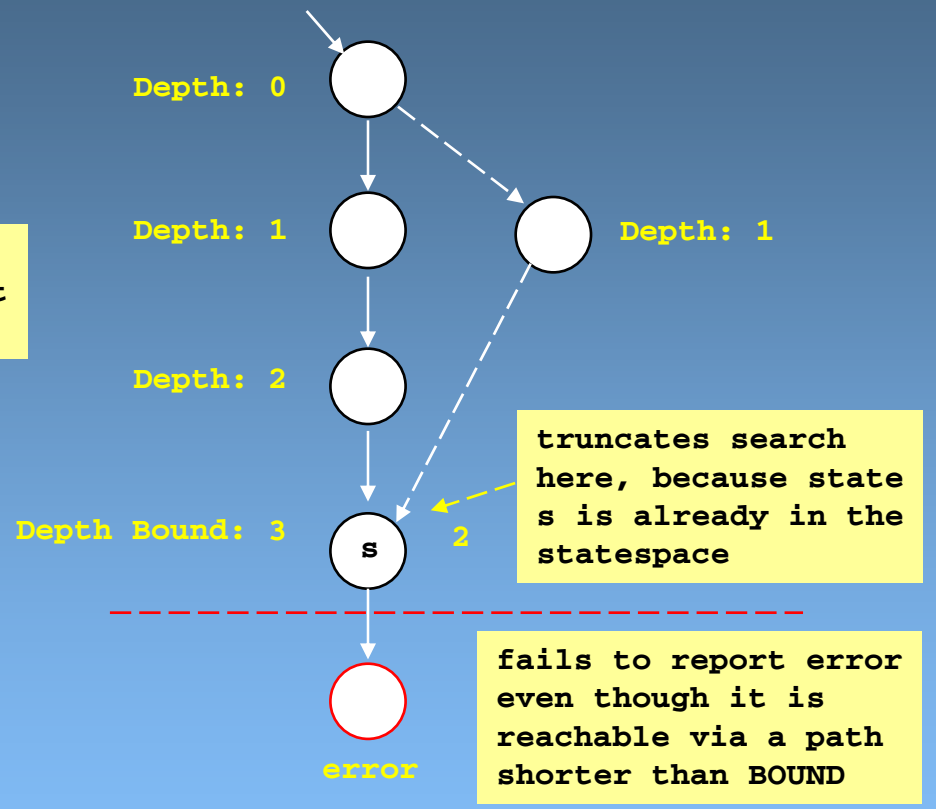
this algorithm trades memory for time but by minimizing memory use we've created *excessive* time overhead

depth-limited search

first try

```
Automaton A = { S, s0, L, T, F }
Stack D = {}
Statespace V = {}
int Depth = 0

Start()
{    Add_Statespace(V, A.s0,0)
     Push_Stack(D, A.s0 )
     Search()
}

Search()
{    if (Depth > BOUND)
           return
     Depth++
     s = Top_Stack(D)
     if (!Safety(s))
           Print_Stack(D)

     for each (s,l,s')∈ A.T
           if In_Statespace(V, s')== false
           {     Add_Statespace(V, s')
                 Push_Stack(D, s')
                 Search()
           }
     Pop_Stack(D)
     Depth--
}
```

adding just this constraint is not enough



depth-limited search

(defined when compiling pan.c with -DREACH)

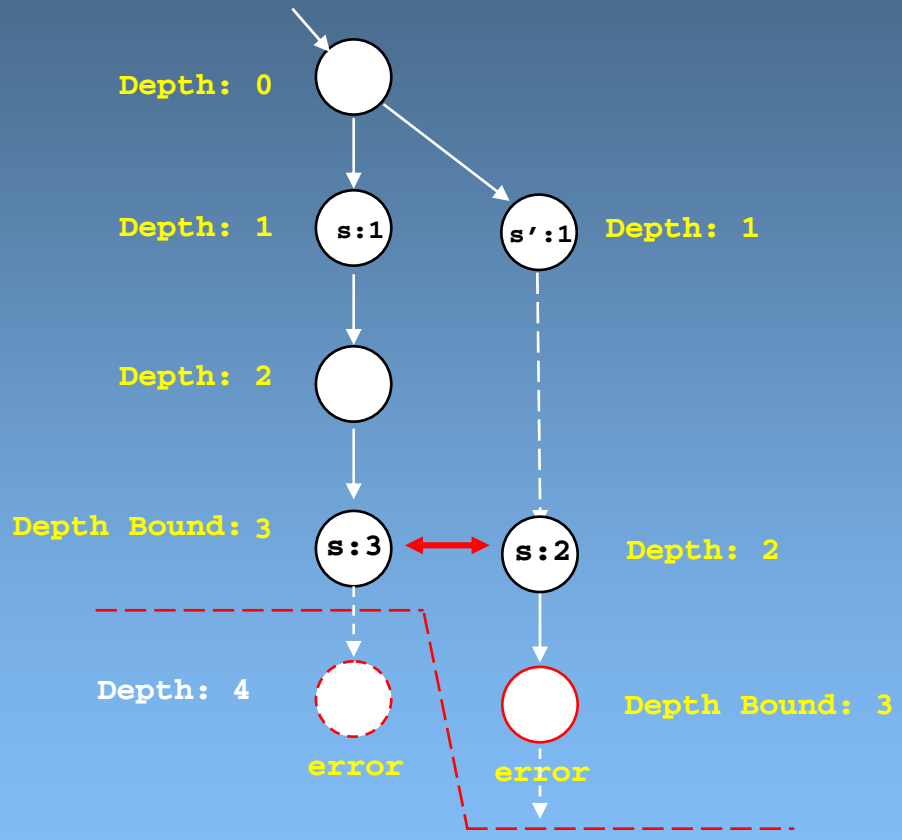
```
Automaton A = { S, s0, L, T, F }
Stack D = {}
Statespace V = {}
int Depth = 0

Start()
{   Add_Statespace(V, A.s0,0)
    Push_Stack(D, A.s0 )
    Search()
}

Search()
{   if (Depth > BOUND)
          return
    Depth++
    s = Top_Stack(D)
    if (!Safety(s))
          Print_Stack(D)

    for each (s,l,s')∈ A.T
          if In_Statespace(V, s', Depth)== false
          {    Add_Statespace(V, s', Depth)
               Push_Stack(D, s')
               Search()
          }
    Pop_Stack(D)
    Depth--
}
```

store the min-depth with each state

```
In_Statespace(V,s,d)

returns false if there is no
state (s',d') in V with s=s'

returns true if there is a
previously stored state (s',d') in V
with s=s' and d >= d'

returns false if there is a
previously stored state (s',d') in V
with s=s' and d < d'
and simultaneously replaces (s',d')
with (s',d)
```

the complexity increases:
in the worst case: if R is the nr of states,
we may explore each state up to R times
(quadratic time complexity in R)
memory use increase only linearly
(to store the depth field)

revised algorithm



breadth-first search

```
Automaton A = { S, s0, L, T, F }
Queue D = {}
Statespace V = {}

Start()
{
   Add_Statespace(V, A.s0)
   Add_Queue(D, A.s0)
   Search()
}

Search()
{
   while (!Empty_Queue(D))
   {   s = Del_Queue(D)
       for each (s,l,s') ∈  A.T
          if In_Statespace(V, s') == false
          {   Add_Statespace(V, s')
              Add_Queue(D, s')
   }      }
}
```

the Stack becomes a FIFO Queue

```
Add_Queue(D,s)
            adds s to ordered set D

Del_Queue(D)
            removes and returns bottom
            element from D

Empty_Queue(D)
            returns true if D contains
            at least one element, and
            otherwise returns false
```

what about trapping safety violations?

(we have no Stack to reproduce the counter-example here...)

NB: Fig. 8.6 in book has an incorrect version of this algorithm

Figure 8.6

Spin's breadth-first search option
(defined by compiling pan.c with –DBFS)

```
Automaton A = { S, s0, L, T, F }
Queue D = {}
Statespace V = {}

Start()
{
   Add_Statespace(V, A.s0, nil)
   Add_Queue(D, A.s0)
   Search()
}

Search()
{
   while (!Empty_Queue(D))
   {    s = Del_Queue(D)

        if (!Safety(s))
           PrintPath(s)

        for each (s,l,s') ∈  A.T
           if In_Statespace(V, s') == s'
           {          Add_Statespace(V, s', s)
                      Add_Queue(D, s')
   }       }
}
```

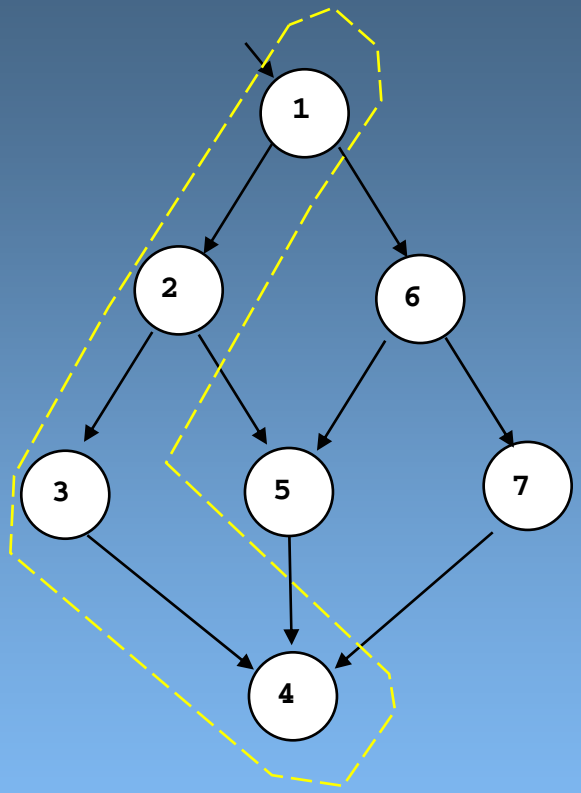
```
Add_Statespace(V,s,s')
   adds state s to set V, together with
   (a pointer to) a predecessor state s'

In_Statespace(V,s)
   returns s if s is not yet in V
   else returns predecessor state s' if
   any, or nil if s has no predecessor
```

```
PrintPath(s)
{  State s' = In_Statespace(V,s);

   if (s' != nil && s' != s)
          PrintPath(s')

   PrintState(s)
}
```

(pointer to) predecessor state s to allow constructing a path from the initial system state to error

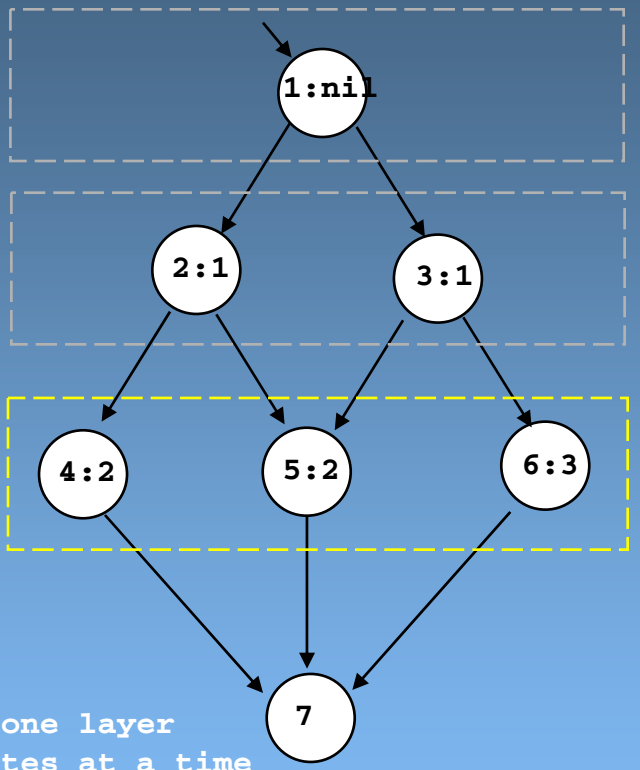
Figure 8.6

dfs and bfs search orders



the stack holds a
path from the root state
towards a leaf state

the statespace of previously
visited states acts as a
search optimization strategy



the queue holds one layer
of successor states at a time

the statespace holds previously
visited states *plus* (pointers to)
predecessor states

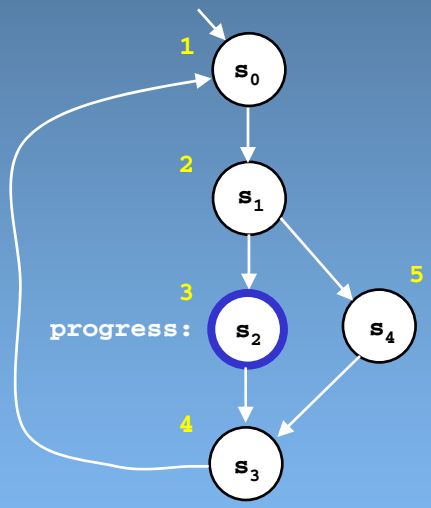
the statespace is now *required* to assure
termination of the search

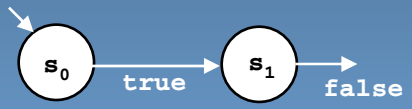
comparing dfs and bfs

- pro:
  - with the breadth-first search, safety violations are detected at the shortest possible distance from the root
- con:
  - in breadth-first search, we can no longer use the contents of the stack to produce a complete counter-example when a safety violation is found
    - we must now store with each state a pointer to at least one predecessor state to be able to reconstruct the path from root to error state, which increases memory use
  - the statespace cannot be lossy
  - no efficient strategy is known for extending a breadth-first search to do cycle detection (to check liveness properties)

liveness properties

- a relatively simple case first:
  - detecting the presence of non-progress cycles



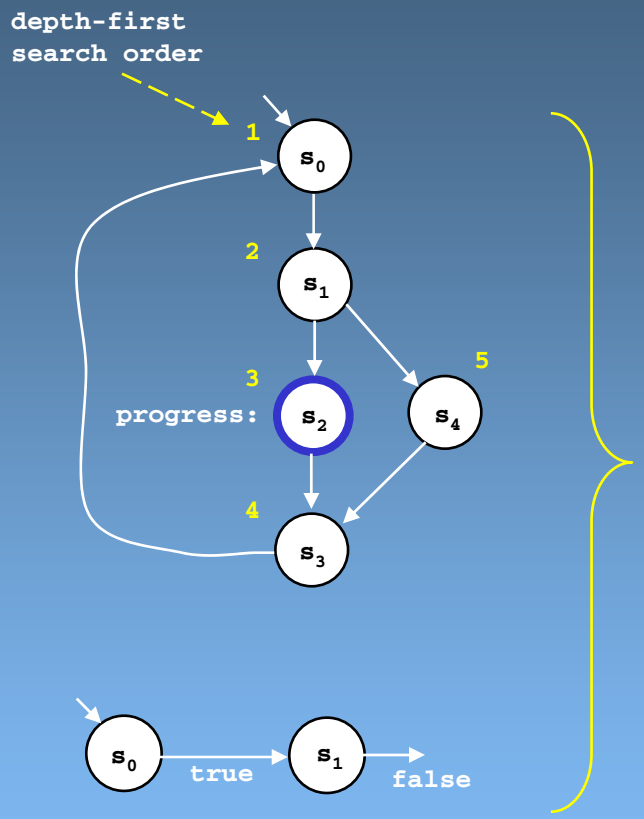


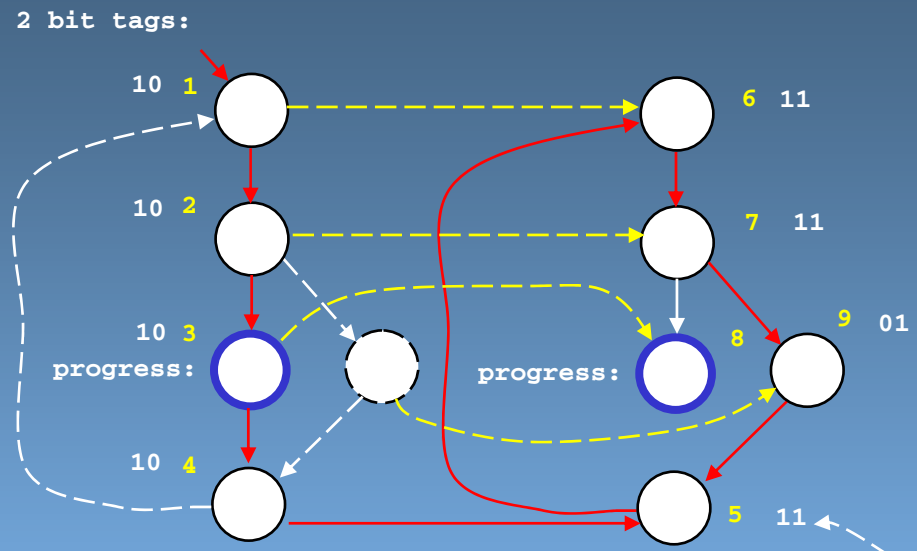
method:
add a 2-state asynchronous demon process that can switch from its initial state s_0 to its final state s_1 at any moment; it cannot exit from its final state

perform the normal depth-first search, with this demon process added, with one change: *block* all transitions that exit a global progress states whenever the demon is in state s_1

***any* cycle in the resulting graph with the demon process in state s_1 is a non-progress cycle**

np detection algorithm
(in early versions of Spin)





a kind of "nested"
depth-first search

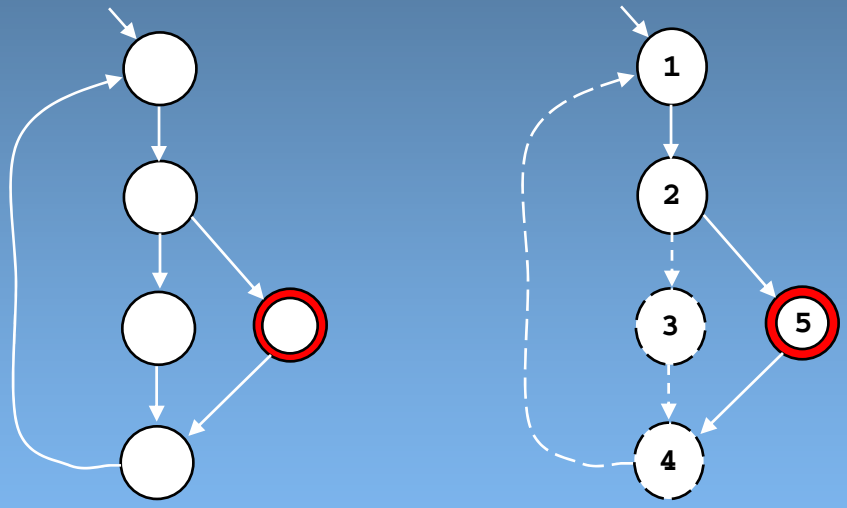
cost:
at most 2x the complexity of a regular depth-first search
the cost is in time, *not* in memory, because we can encode
the state of the demon in 2 bits and add it to
the state-vector; the bits encode for each state whether it
was visited with the demon in state s_0 and/or in state s_1

simple to prove:
if at least one non-progress cycle exists,
this algorithm will always find at least one

the complete execution can be reproduced from the stack
contents during the search, as before

checking omega acceptance
(and general liveness properties)

- to prove liveness properties, it is sufficient if we can detect the *existence* of *cyclic paths* in the product automaton (i.e., the reachability graph) that contain at least one *accepting state*

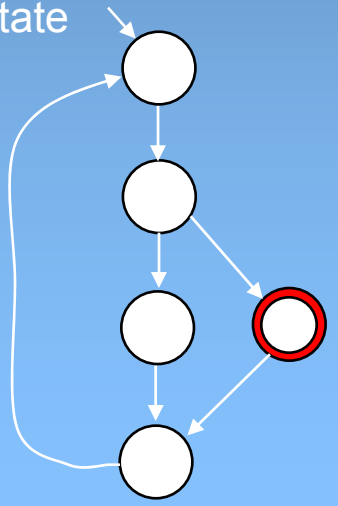


- note that a straight dfs alone does not suffice to detect the acceptance cycle in this graph
  - when revisiting state 4 from 5, we cannot tell that there exists a path from 4 back to 5 (no transitions are stored in the dfs statespace)

different ways of solving this problem

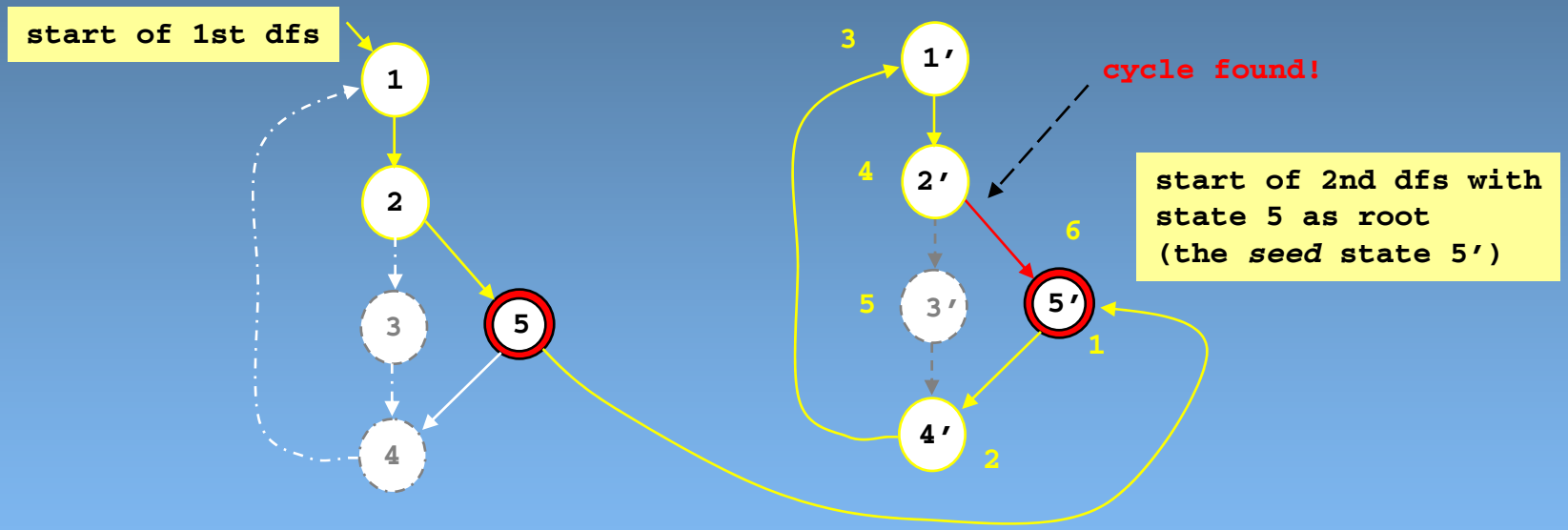
- the default graph search algorithm would be
  - Robert Tarjan's classic depth-first search algorithm
    - computes all *strongly connected components* of a graph
    - a strongly connected component (SCC) is a subset of the states such that each of the states within this subset is reachable from all others in the subset
    - Tarjan's algorithm uses two 32-bit integers per state: the depth-first number and a lowlink number
  - we can check each SCC for the presence of accepting states
    - if found, compute a path to the accepting state
    - and a cyclic path within the SCC

this graph has one SCC, which contains all states



Spin's nested depth-first search algorithm

- to solve the problem in our case, it suffices to know if:
  - there exists *at least one* accepting state that is reachable from the root of the tree AND that is also reachable *from itself*



  - two simpler sub-problems that can each be solved with a simple depth-first search, not requiring the computation of SCCs
  - by combining the two stacks, we again immediately have a complete execution trace for a counter-example