UTA

Today's Agenda

- Quiz 5 (end of the class)
- Quick Review
- Finish Search Algorithms

UTA

Quick Review

- How to check a safety property?

UTA

Search Algorithms

- Introduction
- Computing Automata Product
- Checking Safety Properties
- Checking Liveness Properties
- Search Optimization
- Summary

UTA

It is all about search!

Model checking is basically a process that searches through the global state space.

Checking a safety property is to search for a bad state which violates the property.

Checking a liveness property is to search for a cycle that does not pass through a progress state.

UTA

The challenge

The idea of searching is simple, especially many search algorithms have been developed.

The main challenge with model checking is dealing with the state explosion problem.

The strength of Spin is in its ability to highly optimize the search process.

UTA

Search Algorithms

- Introduction
- Computing Automata Product
- Checking Safety Properties
- Checking Liveness Properties
- Search Optimization
- Summary

UTA

Automata Product

- consider a system of n process, modeled as n finite state automata, A1, A2, …, An
- add property automaton B (derived from an LTL formula)
- the model checker computes the product of these automata: $S = B \otimes \Pi\ Ai$

a synchronous product

an asynchronous product

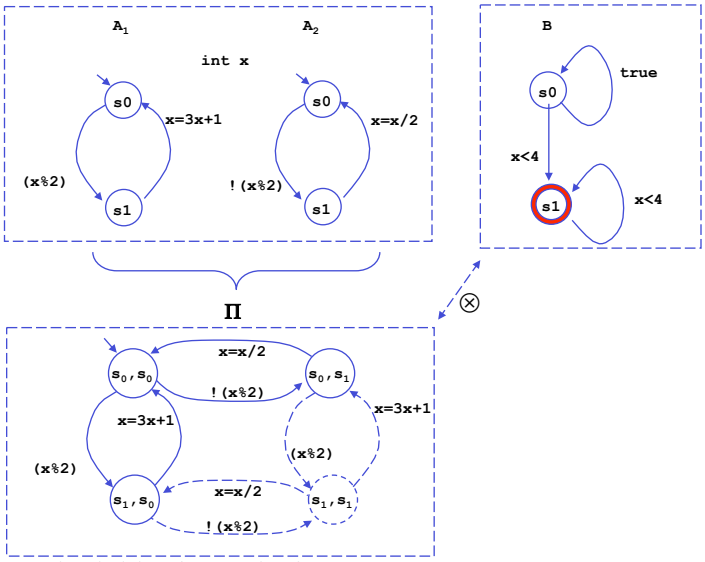
UTA

Example (1)

```
# define p x >= 4
never {    /* ! []<> p */
T0_init:
      if
      :: (! ((p))) -> goto accept_S4
      :: (1) -> goto T0_init
      fi;
accept_S4:
      if
      :: (! ((p))) -> goto accept_S4
      fi;
}

int x = 4;
active proctype A ()
{
    do
    :: x % 2 -> x = 3 *x + 1
    od
}
active proctype B ()
{
    do
    :: ! (x % 2) -> x = x / 2
    od
}
```

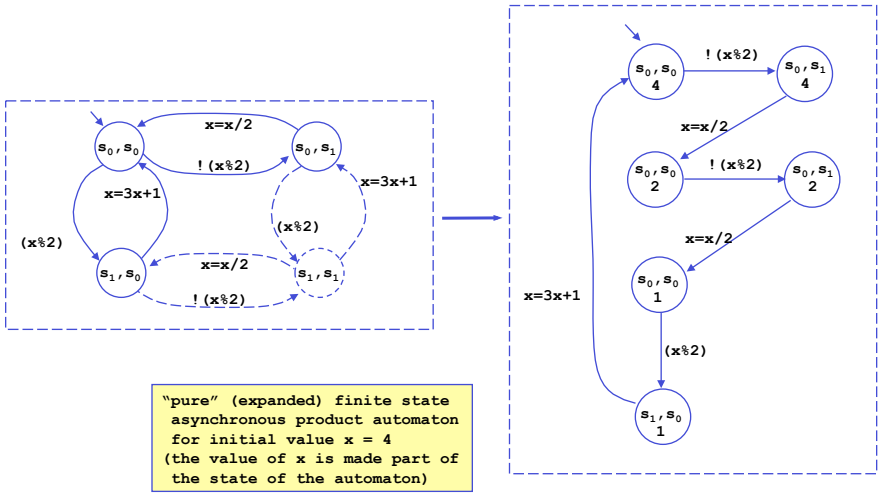
UTA

Example (2)



UTA

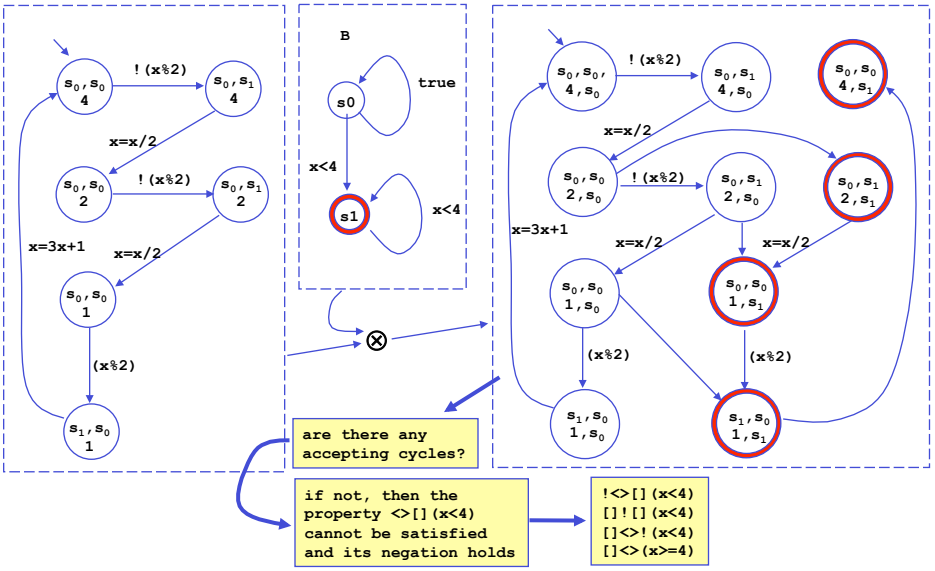
Example (3)



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

UTA

Example (4)



UTA

Search Algorithms

- Introduction
- Computing Automata Product
- Checking Safety Properties
- Checking Liveness Properties
- Search Optimization
- Summary

UTA

Basic DFS (1)

```
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
```

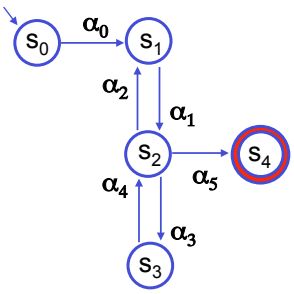
```
Add_Statespace(V,s)
            adds s to set V
In_Statespace(V,s)
            true iff s is in V
```

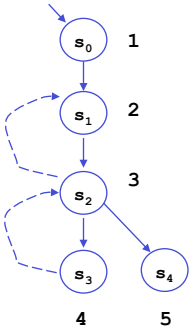
```
the DFS is most easily written
as a recursive procedure -- but the
actual Spin implementation is iterative
to increase efficiency a little
```

Fig. 8.1 p. 168

UTA

Basic DFS (2)





UTA

Basic DFS (3)

```
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)
}
```

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

Fig. 8.2, p. 170

UTA

Stateless Search (1)

no Statespace V

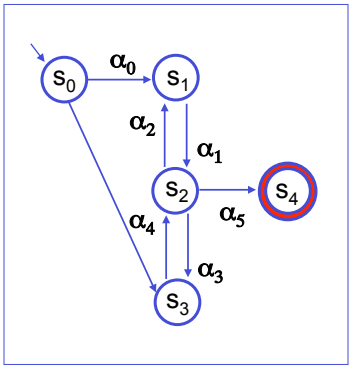
replaced In_Statespace(V,s')
with In_Stack(D,s')

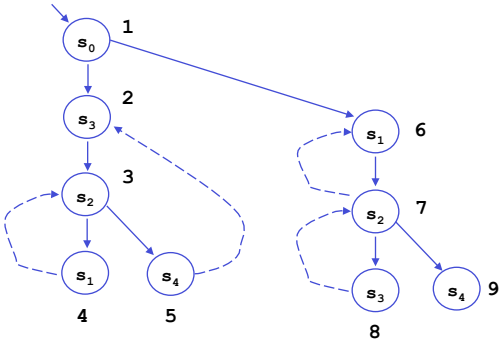
```
Automaton A = { S, s0, L, T, F }
Stack D = {}
/* Statespace V = {} */

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)
}
```

Fig. 8.5 p. 176

UTA

Stateless Search (2)





if s_3 has a sub-tree of 100,000 states the stateless search would visit that entire subtree at least 2 times...

this version of the search visits 9 instead of 5 states... (doing redundant work)
s_3 is visited 2 times here

UTA

BFS (1)

```
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()
{
   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')
                     Search()
          }
}
```

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

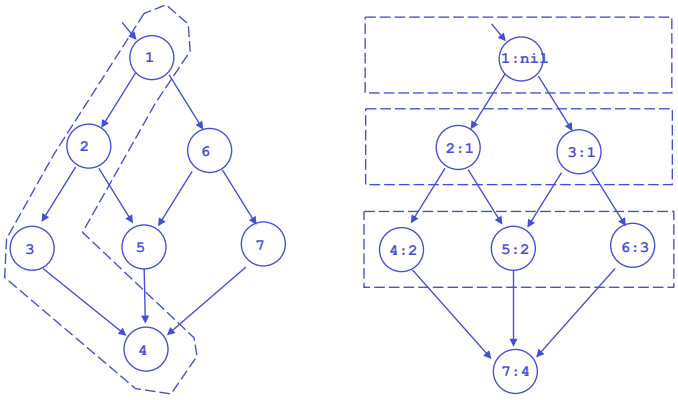
```
PrintPath(s)
{  State s' = In_Statespace(V,s);

   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

UTA

BFS (2)



UTA

BFS (3)

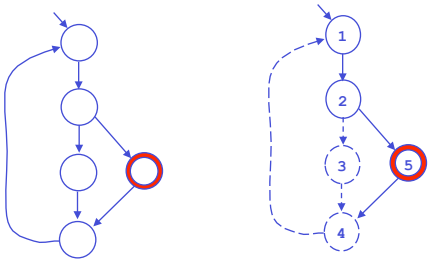
- BFS detects safety violations at the shortest possible path
- Difficult to produce a counter example
- full statespace must be stored to guarantee termination
- no efficient strategy is known for cycle detection (to check liveness properties)

Search Algorithms

- Introduction
- Computing Automata Product
- Checking Safety Properties
- Checking Liveness Properties
- Search Optimization
- Summary

Cyclic Paths

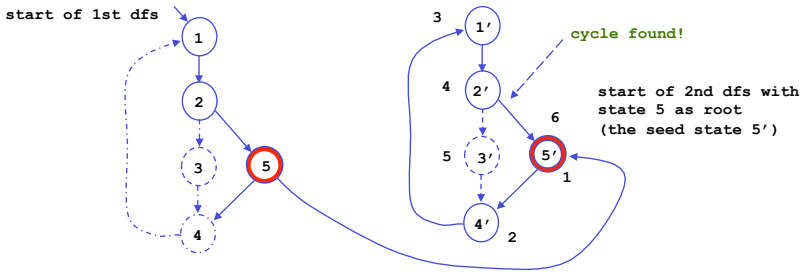
Basically, to prove liveness properties, it is sufficient to show the existence of cyclic paths in the product automaton that contain at least one accepting state.



UTA

Nested Search (1)

The problem is equivalent to show that there exists at least one accepting state that is reachable from the root of the tree AND that is also reachable from itself.



UTA

Nested Search (2)

```
Stack D = {}
Statespace V = {}
State seed = nil
Boolean toggle = false

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

Search()
{
   (s, toggle) = Top_Stack (D)
   for each (s,l,s')∈ A.T
   {       /* check if seed is reachable from itself */
           if (s' == seed ∨ On_Stack (D, s', false)
           {   PrintStack (D)
               PopStack (D)
               return
           }
           if In_Statespace(V, s', toggle)== false
                 {   Add_Statespace(V, s', toggle)
                     Push_Stack(D, s', toggle)
                     Search()
                 }
   }
   if s ∈ A.F && toggle == false
   {       seed = s       /* reachable accepting state */./
           toggle = true
           Push_Stack (D, s, toggle)
           Search ()      /* start 2nd search */.
           Pop_Stack (D)
           seed = nil
           toggle = false
   }

   Pop_Stack (D)
}
```

UTA

Search Algorithms

- Introduction
- Computing Automata Product
- Checking Safety Properties
- Checking Liveness Properties
- Search Optimization
- Summary

UTA

State Explosion Problem

The number of reachable states in a system is an exponential function of the number of processes in the system.

The main challenge of model checking is dealing with this state explosion problem.

UTA

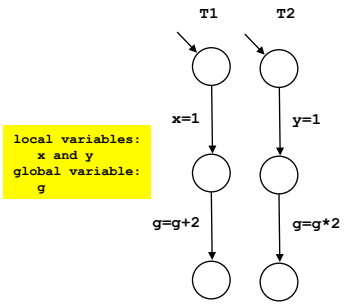
Optimization Techniques

There are two types of optimization techniques employed in Spin:

- techniques to reduce the number of states that needs to be searched
- techniques to efficiently store the states in memory

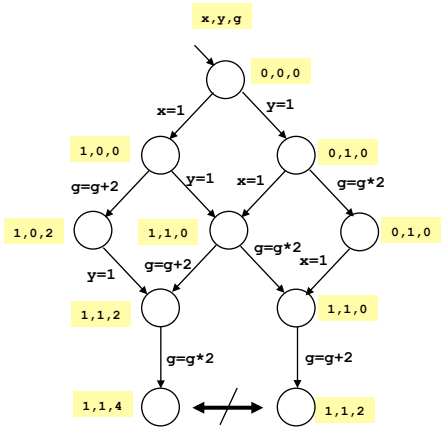
UTA

Example (1)



six runs:

```
x=1;g=g+2;y=1;g=g*2
x=1;y=1;g=g+2;g=g*2
x=1;y=1;g=g*2;g=g+2
y=1;g=g*2;x=1;g=g+2
y=1;x=1;g=g*2;g=g+2
y=1;x=1;g=g+2;g=g*2
```



only two operations share data:
g=g+2 and g=g*2
all other combinations of operations
are data-independent, e.g. x=1 and g=g+2

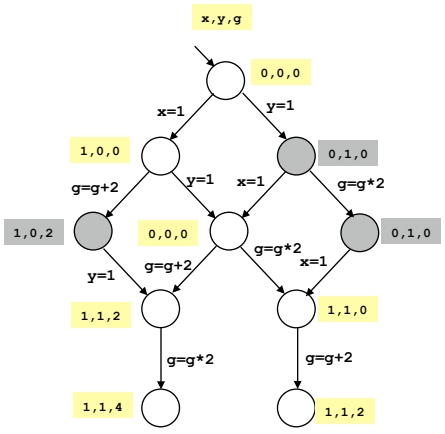
UTA

Example (2)

```
independent pairs:
    x=1 <-> y=1
    x=1 <-> g=g*2
    y=1 <-> g=g+2
```

2 groups of 3 equivalent runs each:

```
x=1;g=g+2;y=1;g=g*2
x=1;y=1;g=g+2;g=g*2
y=1;x=1;g=g+2;g=g*2

x=1;y=1;g=g*2;g=g+2
y=1;x=1;g=g*2;g=g+2

y=1;g=g*2;x=1;g=g+2
```



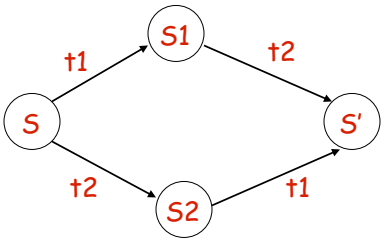
reducing R from 10 to 7 states

UTA

Independence

- Two transitions t1 and t2 are independent if the following two conditions are met:
  - Enabledness: The execution of t1 does not disable t2, and vice versa.
  - Commutativity: The execution of t1 and t2 in any order leads to the same state.
- Two transitions t1 and t2 are dependent if they are not independent

Example



Visibility

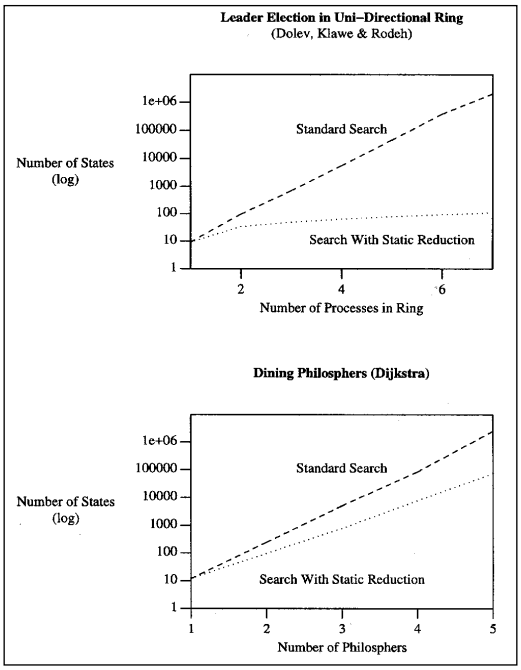
- A transition is invisible if its execution does not change the value of the property being checked. A transition is visible if it is not invisible.
- If t1 and t2 are invisible transitions, then it is safe to ignore state s2.

UTA

Effect of POR

best case →

worst case →



UTA

State Compression - Basic Idea

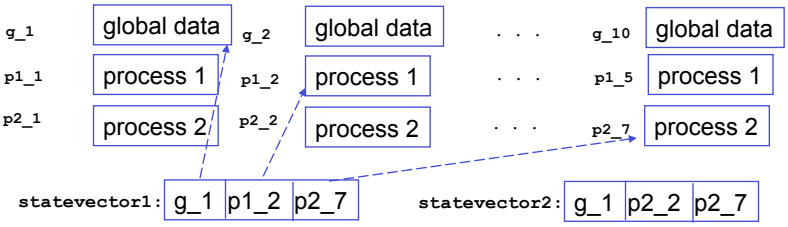
The basic idea is that a small number of local component typically appear in many different global states.

A global state is broken down into separate components: (1) global data; (2) local components (one for each active process).

Each component is stored separately in a lookup table, and is given a unique index-number; only the index numbers are used to form a global state.

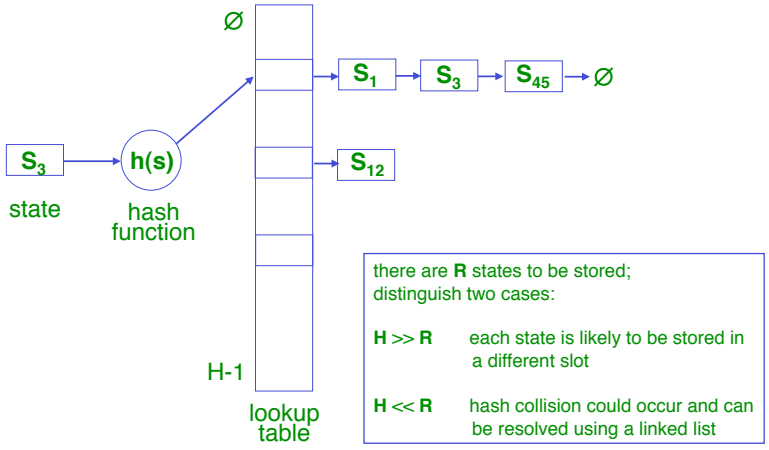
UTA

State Expression - Example



UTA

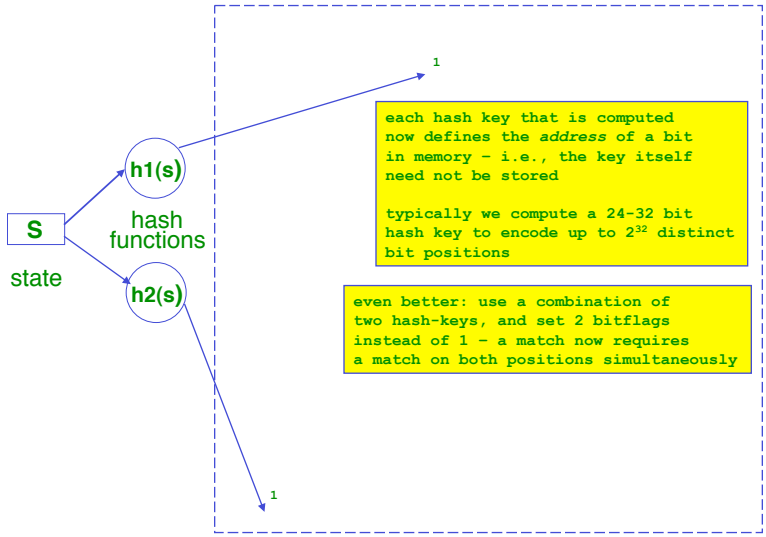
Hash Table Lookup



Bithash

- In the case where H >> R there is no need to store the states.
- the possibility of a hash-collision now becomes remote
- trading increased memory use for increased accuracy:
  - instead of 1 hash-function, use k>1 independent hash-functions
  - "store" each state k times
  - a hash-collision now requires k matches

Bitstate Array



UTA

Search Algorithms

- Introduction
- Computing Automata Product
- Checking Safety Properties
- Checking Liveness Properties
- Search Optimization
- Summary

UTA

Summary

- DFS is usually preferred over BFS for model checking. Why?
- Liveness properties are more expensive to check than safety properties. Why?
- Partial order reduction conducts a selective search to reduce the number of states to be searched.
- State compression does not reduce the number of states, but reduces the amount of memory to store them.
- Bitstate hashing further reduces the memory requirement, but is subject to incomplete coverage.