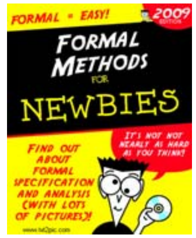
CISC422/853: Formal Methods in Software Engineering: Computer-Aided Verification



Topic 5: Model Checking, Part 1

Juergen Dingel
Feb, 2009

Readings: Spin book

- Chapter 11 (Using Spin)
- Chapter 8, pages 167-178 (Search Algorithms)

Outline

- The SumToN Example (Source: CIS842 @ KSU)
- Use this simple example to explain
  - schedules
  - computation trees
  - 3 forms of exploration:
    - random
    - interactive
    - exhaustive

SumToN

```
system SumToN {
 const PARAM { N = 1 };
 typealias byte int wrap (0,255);

 byte x := 1;
 byte t1;
 byte t2;

 active thread Thread1() {
  loc loc0:
   when x != (byte)0 do { t1 := x; }
   goto loc1;

  loc loc1:
   do { t2 := x; }
   goto loc2;

  loc loc2:
   do { x := t1 + t2; }
   goto loc0;
 }
```

```
 active thread Thread2() {
  loc loc0:
   when x != (byte)0 do { t1 := x; }
   goto loc1;

  loc loc1:
   do { t2 := x; }
   goto loc2;

  loc loc2:
   do { x := t1 + t2; }
   goto loc0;
 }

 active thread Thread0() {
  loc loc0:
   do { assert (x != (byte)PARAM.N); }
   return;
 }
}
```

Source: 842@KSU

SumToN (Cont'd)

```
system SumToN {
 const PARAM { N = 1 };
 typealias byte int wrap (0,255);

 byte x := 1;
 byte t1;
 byte t2;

 active thread Thread1() {
  loc loc0:
   when x != (byte)0 do { t1 := x; }
   goto loc1;

  loc loc1:
   do { t2 := x; }
   goto loc2;

  loc loc2:
   do { x := t1 + t2; }
   goto loc0;
 }
```

declare a 'byte' to be an integer with range 0..255 that will 'wrap around' when operated on

```
 active thread Thread2() {
  loc loc0:
   when x != (byte)0 do { t1 := x; }
   goto loc1;
   ...
 }

 active thread Thread0() {
  loc loc0:
   do { assert (x != (byte)PARAM.N); }
   return;
 }
}
```

Source: 842@KSU

SumToN (Cont'd)

```
system SumToN {
 const PARAM { N = 1 };
 typealias byte int wrap (0,255);

 byte x := 1;
 byte t1;
 byte t2;

 active thread Thread1() {
  loc loc0:
   when x != (byte)0 do { t1 := x; }
   goto loc1;

  loc loc1:
   do { t2 := x; }
   goto loc2;

  loc loc2:
   do { x := t1 + t2; }
   goto loc0;
 }
```

declare three byte-sized variables

```
 active thread Thread2() {
  loc loc0:
   when x != (byte)0 do { t1 := x; }
   goto loc1;

  loc loc2:
   do { x := t1 + t2; }
   goto loc0;
 }

 active thread Thread0() {
  loc loc0:
   do { assert (x != (byte)PARAM.N); }
   return;
 }
}
```

Source: 842@KSU

SumToN (Cont'd)

```
system SumToN {
 const PARAM { N = 1 };
 typealias byte int wrap (0,255);

 byte x := 1;
 byte t1;
 byte t2;

 active thread Thread1() {
  loc loc0:
   when x != (byte)0 do { t1 := x; }
   goto loc1;

  loc loc1:
   do { t2 := x; }
   goto loc2;

  loc loc2:
   do { x := t1 + t2; }
   goto loc0;
 }
```

```
 active thread Thread2() {
  loc loc0:
   when x != (byte)0 do { t1 := x; }
   goto loc1;

  loc loc1:
   do { t2 := x; }
   goto loc2;

  loc loc2:
   do { x := t1 + t2; }
   goto loc0;
 }

 ...read0() {
  ...!= (byte)PARAM.N); }
```

Each thread reads the value of x in t1, then t2, then sums t1 and t2 to get a new value for x

Source: 842@KSU

SumToN (Cont'd)

```
system SumToN {
 const PARAM { N = 1 };
 typealias byte int wrap (0,255);

 byte x := 1;
 byte t1;
 byte t2;

 active thread Thread1() {
  loc loc0:
   when x != (byte)0 do { t1 := x; }
   goto loc1;

  loc loc1:
   do { t2 := x; }
   goto loc2;

  loc loc2:
   do { x := t1 + t2; }
   goto loc0;
 }
```

The "main" thread asserts that x is not equal to the value of N

```
 active thread Thread2() {
  loc loc0:
   when x != (byte)0 do { t1 := x; }
   ...loc1;

  ...1:
   ...2 := x; }
   ...loc2;

  loc loc2:
   do { x := t1 + t2; }
   goto loc0;
 }

 active thread Thread0() {
  loc loc0:
   do { assert (x != (byte)PARAM.N); }
   return;
 }
}
```

Source: 842@KSU

SumToN

```
system SumToN {
 const PARAM { N = 1 };
 typealias byte int wrap (0,255);

 byte x := 1;
 byte t1;
 byte t2;

 active threa...
  loc loc0:
   when x !=...
   goto loc1;...

  loc loc1:
   do { t2 :=...
   goto loc2;...

  loc loc2:
   do { x := t1 + t2; }
   goto loc0;
 }
```

- This transition can be arbitrarily interleaved with all others from Thread1 and Thread2.
- This is how we check invariants (properties that should always hold)

```
 active thread Thread2() {
  loc loc0:
   when x != (byte)0 do { t1 := x; }
   goto loc1;

  loc loc1:
   do { t2 := x; }
   goto loc2;

  loc loc2:
   do { x := t1 + t2; }
   goto loc0;
 }

 active thread Thread0() {
  loc loc0:
   do { assert (x != (byte)PARAM.N); }
   return;
 }
}
```

Source: 842@KSU

10^6 $ Question

Can the assertion in the SumToN example be violated? If so, how?

- Answering this question requires us to reason about possible *schedules* (i.e., orderings of instruction execution)
- Let's try to find schedules that cause the assertion to be violated for various values of N…

SumToN Assertion Violation

```
     active thread Threadk() {
       loc loc0:
k:0      when x != (byte)0 do {
             t1 := x; }
         goto loc1;

k:1    loc loc1:
         do { t2 := x; }
         goto loc2;

k:2    loc loc2:
         do { x := t1 + t2; }
         goto loc0;
     }

0:0  active thread Thread0() {
       loc loc0:
         do { assert (x !=
               (byte)PARAM.N); }
         return;
     }
   }
```

Violating schedule for N = 1:

(initial state) [0, 0, 0, x = 1, t1 = 0, t2 = 0]

0:0 → [-, 0, 0, x = 1, t1 = 0, t2 = 0] — violation! (x = 1)

…that was easy!

Source: 842@KSU

SumToN Assertion Violation (Cont'd)

```
     active thread Threadk() {
       loc loc0:
k:0      when x != (byte)0 do {
             t1 := x; }
         goto loc1;

k:1    loc loc1:
         do { t2 := x; }
         goto loc2;

k:2    loc loc2:
         do { x := t1 + t2; }
         goto loc0;
     }

0:0  active thread Thread0() {
       loc loc0:
         do { assert (x !=
               (byte)PARAM.N); }
         return;
     }
   }
```

1st violating schedule for N = 2:

(initial state) [0, 0, 0, x = 1, t1 = 0, t2 = 0]

1:0 → [0, 1, 0, x = 1, t1 = 1, t2 = 0]

1:1 → [0, 2, 0, x = 1, t1 = 1, t2 = 1]

1:2 → [0, 0, 0, x = 2, t1 = 1, t2 = 1]

0:0 → [-, 0, 0, x = 2, t1 = 1, t2 = 1] — violation! (x = 2)

Source: 842@KSU

SumToN Assertion Violation (Cont'd)

```
     active thread Threadk() {
       loc loc0:
k:0      when x != (byte)0 do {
             t1 := x; }
         goto loc1;

k:1    loc loc1:
         do { t2 := x; }
         goto loc2;

k:2    loc loc2:
         do { x := t1 + t2; }
         goto loc0;
     }

0:0  active thread Thread0() {
       loc loc0:
         do { assert (x !=
               (byte)PARAM.N); }
         return;
     }
   }
```

2nd violating schedule for N = 2:

(initial state) [0, 0, 0, x = 1, t1 = 0, t2 = 0]

2:0 → [0, 0, 1, x = 1, t1 = 1, t2 = 0]

2:1 → [0, 0, 2, x = 1, t1 = 1, t2 = 1]

2:2 → [0, 0, 0, x = 2, t1 = 1, t2 = 1]

0:0 → [-, 0, 0, x = 2, t1 = 1, t2 = 1] — violation! (x = 2)

Source: 842@KSU

SumToN Assertion Violation (Cont'd)

```
k:0  active thread Threadk() {
       loc loc0:
         when x != (byte)0 do {
             t1 := x; }
         goto loc1;
k:1    loc loc1:
         do { t2 := x; }
         goto loc2;
k:2    loc loc2:
         do { x := t1 + t2; }
         goto loc0;
     }
0:0  active thread Thread0() {
       loc loc0:
         do { assert (x !=
               (byte)PARAM.N); }
         return;
     }
   }
```

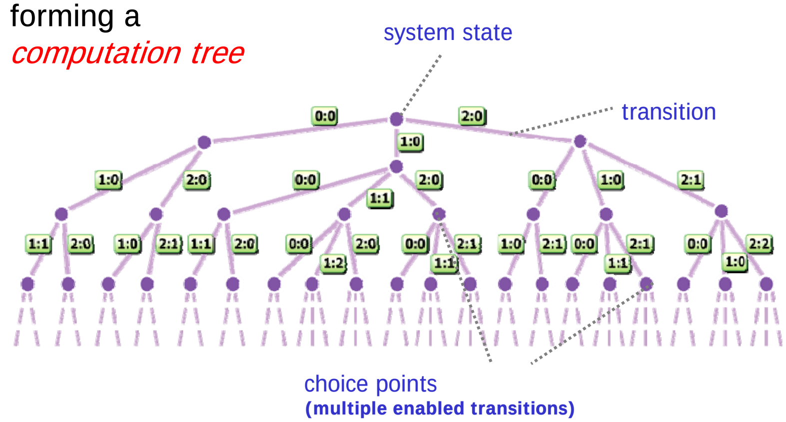
3rd violating schedule for N = 2:

(initial state)	[0, 0, 0, x = 1, t1 = 0, t2 = 0]
1:0	[0, 1, 0, x = 1, t1 = 1, t2 = 0]
2:0	[0, 1, 1, x = 1, t1 = 1, t2 = 0]
2:1	[0, 1, 2, x = 1, t1 = 1, t2 = 1]
2:2	[0, 1, 0, x = 2, t1 = 1, t2 = 1]
0:0	[-, 1, 0, x = 2, t1 = 1, t2 = 1]

violation!

Computation Trees

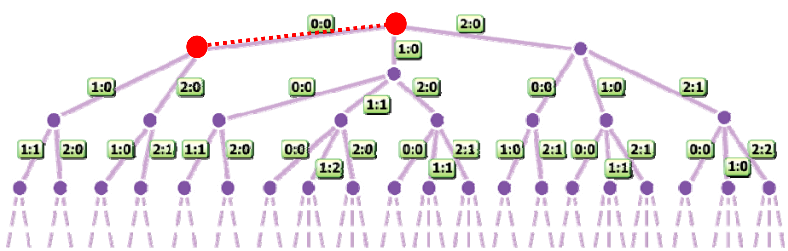
We can think of the possible schedules (execution traces) as forming a *computation tree*



Computation Trees (Cont'd)

We can think of the possible schedules (execution traces) as forming a *computation tree...*

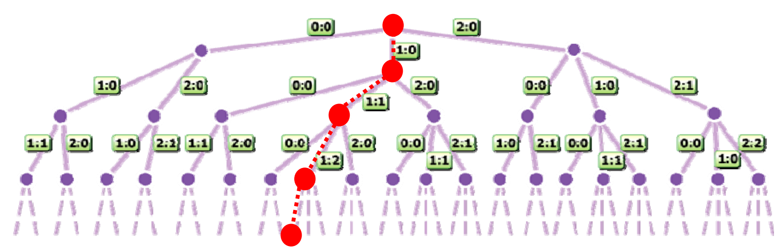
First example trace (schedule)



Computation Trees (Cont'd)

We can think of the possible schedules (execution traces) as forming a *computation tree...*

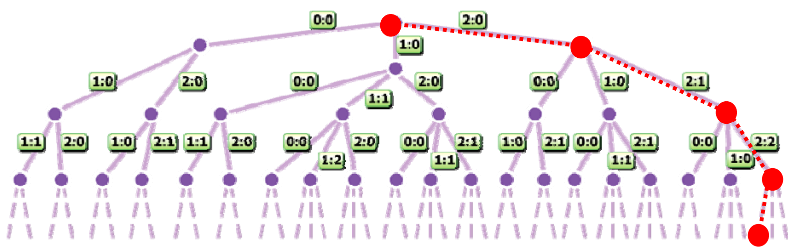
Second example trace (schedule)



Computation Trees (Cont'd)

We can think of the possible schedules (execution traces) as forming a *computation tree...*

Third example trace (schedule)

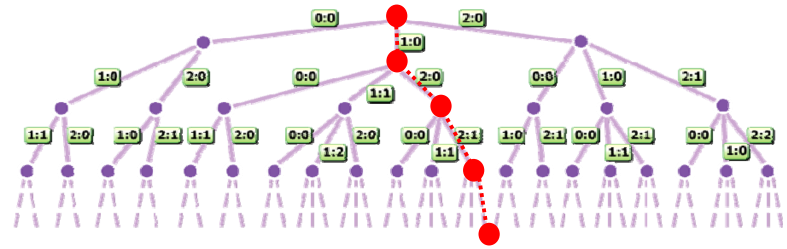


Source: 842@KSU

Computation Trees (Cont'd)

We can think of the possible schedules (execution traces) as forming a *computation tree...*

Fourth example trace (schedule)



Source: 842@KSU

Computation Trees, Formally

Given a FSA A, the *computation tree* T_A of A is obtained by

- s_0 is root of T_A for $s_0 \in A.S_0$
- "unwinding" the tree using $A.\delta$:
  - for every s in T_A, s' is successor of s iff $(s, l, s') \in A.\delta$ for some l

Example:

Observations:

- $paths(T_A) = runs(A)$
- a state may occur more than once along a path in T_A
- states w/o outgoing transitions in A are leaves in T_A
- every path in T_A is infinite iff transition relation $A.\delta$ is total

Aside: Model Checking "On-the-Fly"

- Let
  - D be representation of a system in input language of some model checker MC
  - $iFSA_D$ iFSA/computation tree corresponding to D
- Two kinds of model checkers:
  - On-the-fly: MC computes $iFSA_D$ step-by-step during exploration
    - Examples: Spin, Bogor
  - Not on-the-fly: MC computes $iFSA_D$ before it starts the exploration
    - Example: NuSMV
- What are the pros and cons of "on-the-fly" model checking?

Aside: Model Checking Symbolically

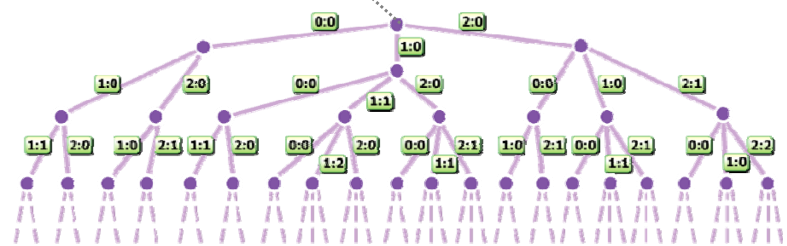
- Model checkers that are not on the fly, typically use a sophisticated data structure called

  Binary Decision Diagrams (BDDs)

  to represent $iFSA_D$
- BDDs represent $iFSA_D$ "symbolically" in a graph rather than explicitly
- For many D, BDDs allow transition relation of $iFSA_D$ to be represented very efficiently (through lots of sharing)
- SMV, Cadence SMV, and NuSMV:
  - BDDs were first used for model checking in SMV (Symbolic Model Verifier, developed at CMU)
  - Cadence SMV: developed at Cadence Labs (for Windows)
  - NuSMV: open-source effort by IRST (Trento, Italy) and CMU

Random Simulation

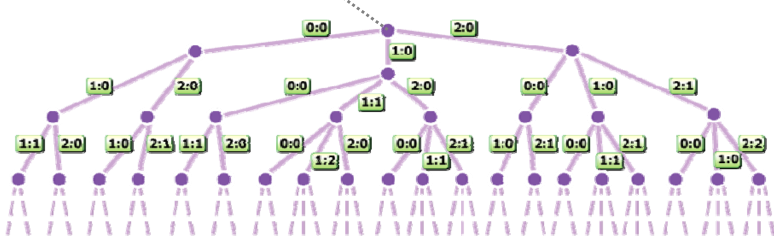
In a random simulation, Bogor randomly chooses a branch at a choice point



Source: 842@KSU

Guided Simulation

In a guided simulation, Bogor asks the user which transition to take at a choice point



Source: 842@KSU

Exhaustive Exploration

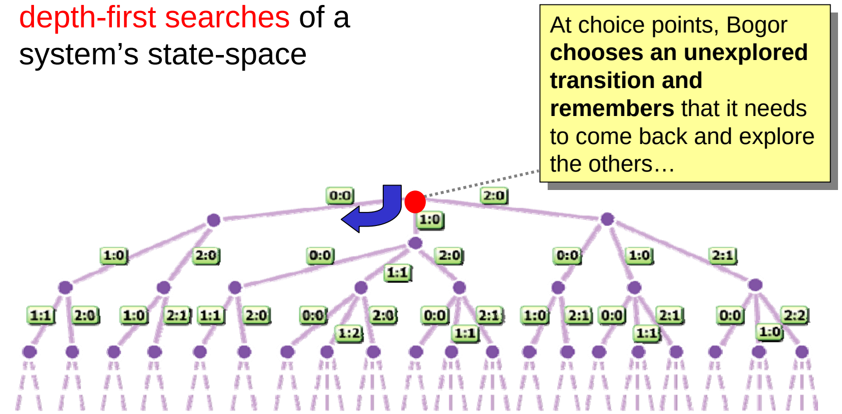
- Both in random and interactive exploration, only one path is explored at a time
- If during random and interactive exploration
  - violation found, then system incorrect (due to soundness)
  - no violation found, then ??
- Little better than a using a good debugger
- We really want exhaustive exploration:
  - Using exhaustive exploration, all executions (schedules) of the system are checked for violations. So, if
    - violation found, then system incorrect (soundness)
    - no violation found, then system correct (completeness)

Exhaustive Exploration (Cont'd)

- Model checkers allow you to perform exhaustive explorations
- Challenge: Exploration may take
  - a long time, because
    - the system has lots of reachable states
    - the system has lots of executions
  - a lot of memory, because
    - states contain lots of information (e.g., processes have lots of variables, or variables range over complex data structures)
- Need safe optimizations (Topic 8)
- But before that, we discuss algorithms for exhaustive exploration. All are based on DFS and BFS
- Using Bogor as example (Spin works similarly)

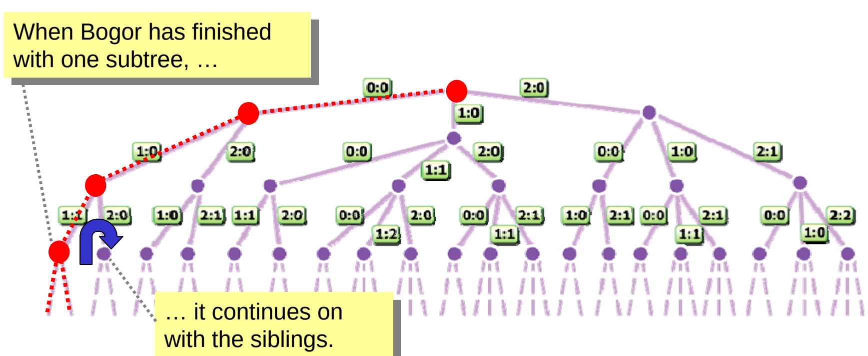
Exhaustive Depth-first Search

Bogor can perform exhaustive depth-first searches of a system's state-space



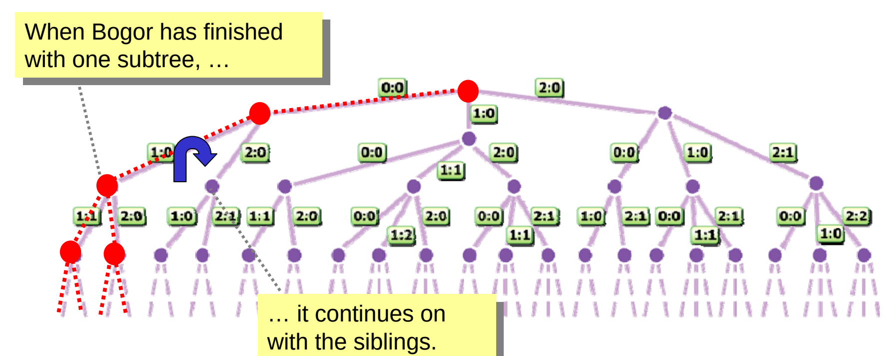
Exhaustive Depth-first Search (Cont'd)

Bogor can perform exhaustive depth-first searches of a system's state-space



Exhaustive Depth-first Search (Cont'd)

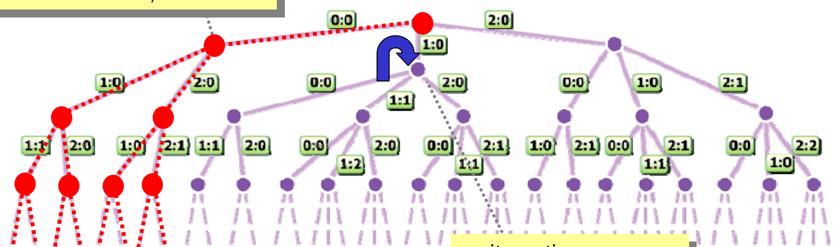
Bogor can perform exhaustive depth-first searches of a system's state-space



Exhaustive Depth-first Search (Cont'd)

Bogor can perform exhaustive depth-first searches of a system's state-space

When Bogor has finished with one subtree, …

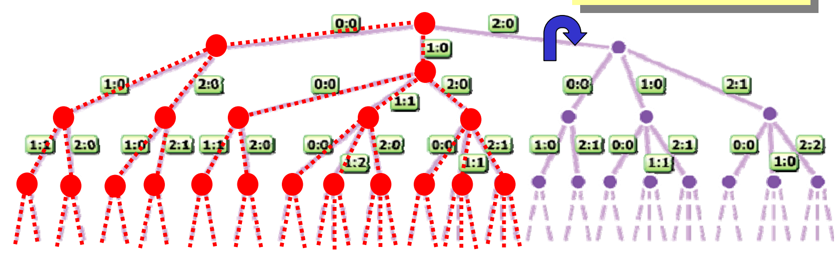


… it continues on with the siblings.

Exhaustive Depth-first Search (Cont'd)

Bogor can perform exhaustive depth-first searches of a system's state-space

… until the entire computation tree is covered.

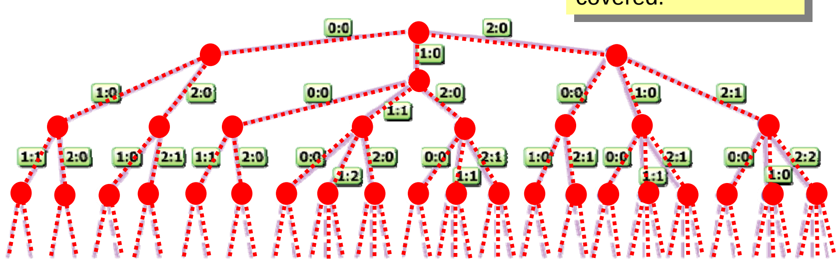


Source: 842@KSU

Exhaustive Depth-first Search (Cont'd)

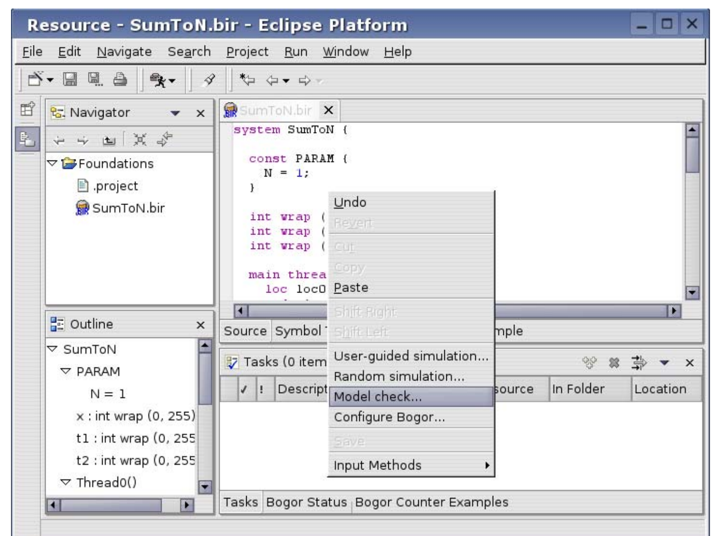
Bogor can perform exhaustive depth-first searches of a system's state-space.

… until the entire computation tree is covered.

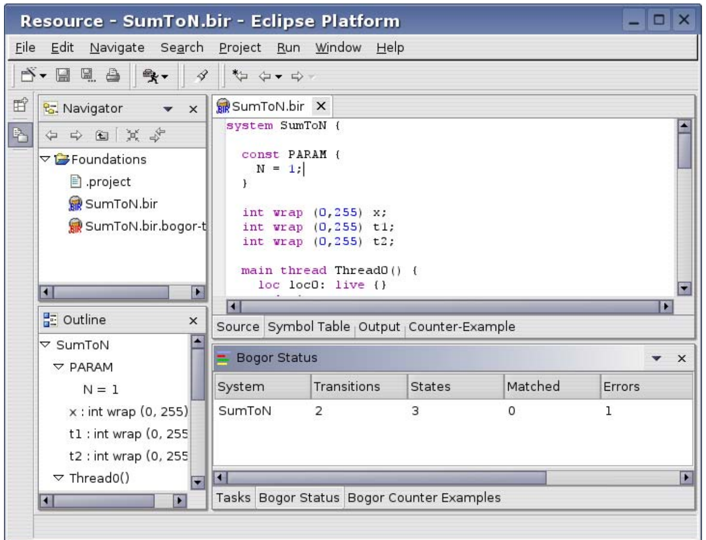


Source: 842@KSU

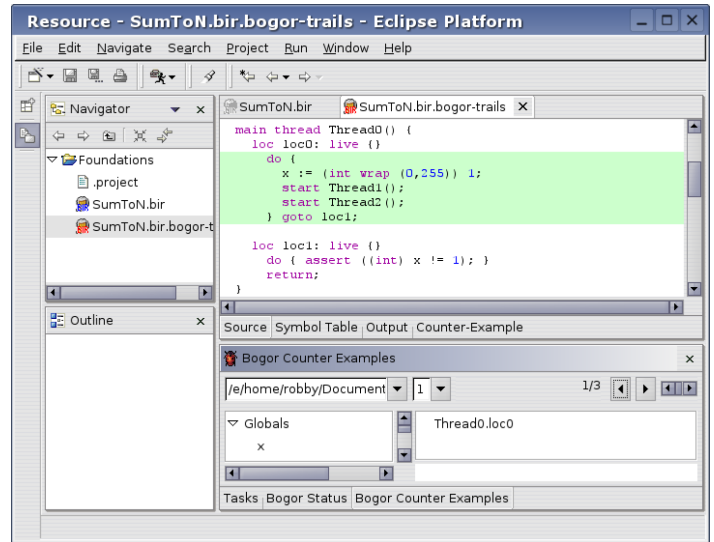
DFS with Bogor



Bogor Output



Bogor Counterexample Display

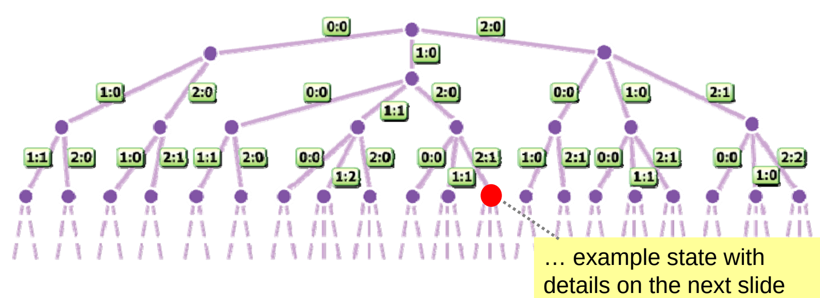


DFS Basic Data Structures

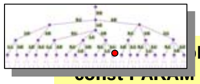
- **State vector**
  - contains values of all variables and program counters for each process
- **Depth-first stack**
  - contains states (or transitions) encountered down a certain path in computation tree
- **Seen state set**
  - contains state vectors for all states that have been checked already (seen) during depth-first search
- **Note**
  - values of these data structures shown in abstract manner only
  - actual implementation of most model-checkers uses multiple clever representations to obtain a highly optimized search algorithm

SumToN State Vector Example

The state vector is the data structure corresponding to the state (as previously discussed). It holds the value of all variables as well as program counters for each process, and represents a particular position in the computation tree



SumToN State Vector Example



```
...N {
const PARAM { N = 1 };
typealias byte int wrap (0,25

byte x := 1;
byte t1;
byte t2;

active thread Thread1() {
  loc loc0:
    when x != (byte)0 do { t1 := x; }
    goto loc1;

  loc loc1:
    do { t2 := x; }
    goto loc2;

  loc loc2:
    do { x := t1 + t2; }
    goto loc0;
}
```

…program counters for each thread

```
active thread Thread2() {
  loc loc0:
    when x != (byte)0 do { t1 := x; }
    goto loc1;

  loc loc1:
    do { t2 := x; }
    goto loc2;

  loc loc2:
    do { x := t1 + t2; }
    goto loc0;
  }

 active thread Thread0() {
  loc loc0:
    do { assert (x != (byte)PARAM.N); }
    return;
  }
}
```

Example State Vector: [0,0,2,1,1,1]

SumToN Assertion Violation

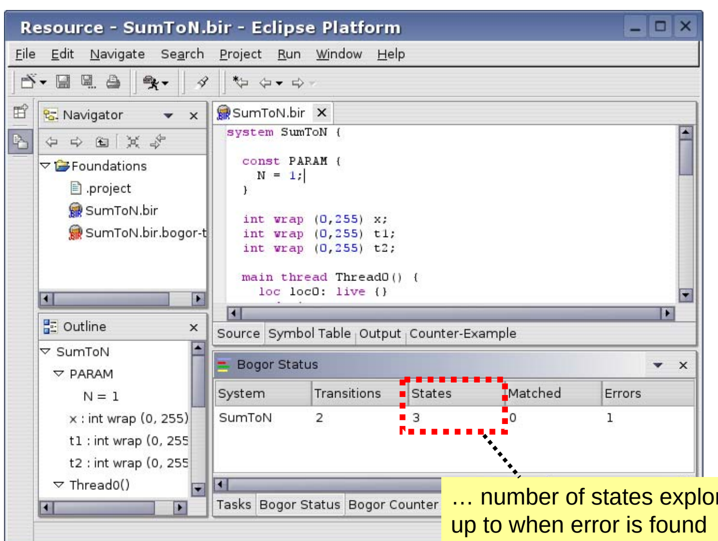
```
active thread Threadk() {
k:0  loc loc0:
       when x != (byte)0 do {
       t1 := x; } goto loc1;

k:1  loc loc1:
       do { t2 := x; }
       goto loc2;

k:2  loc loc2:
       do { x := t1 + t2; }
       goto loc0;
     }

   active thread Thread0() {
0:0  loc loc0:
       do {
         assert (x !=
           (byte)PARAM.N); }
       return;
     }
}
```

Violating schedule for N = 2

(initial values) [0, 0, 0, x = 1, t1 = 0, t2 = 0]
1:0 → [0, 1, 0, x = 1, t1 = 1, t2 = 0]
2:0 → [0, 1, 1, x = 1, t1 = 1, t2 = 0]
2:1 → [0, 1, 2, x = 1, t1 = 1, t2 = 1]
2:2 → [0, 1, 0, x = 2, t1 = 1, t2 = 1]
0:0 → [-, 1, 0, x = 2, t1 = 1, t2 = 1]

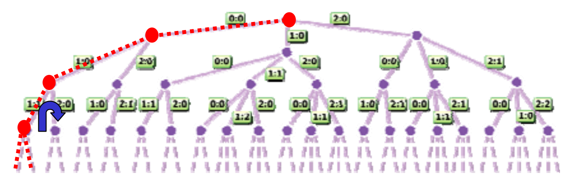
…recall state vectors leading to violation of assertion

Bogor Output

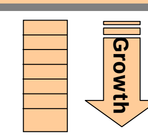


… number of states explored up to when error is found

Depth-First Stack

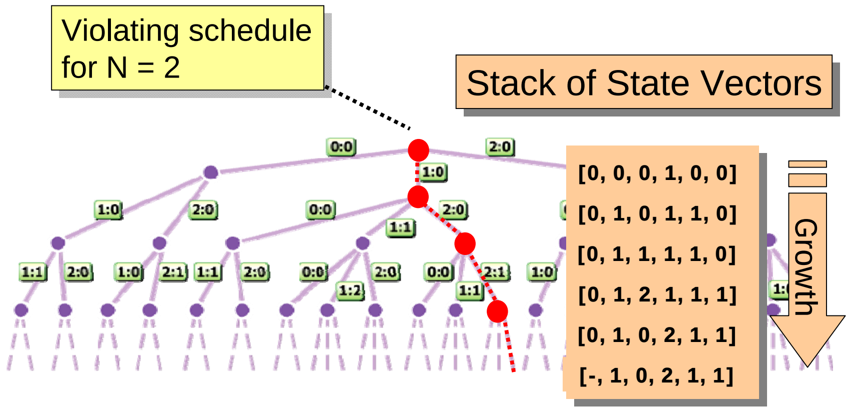


Depth-first Stack



- The depth-first stack serves two purposes:
  - When search comes to end of a path (or a state that has been seen before) and backtrack, the top of stack tells us where to backtrack to
  - If an error is encountered, the current contents of stack gives the computation path that leads to the error (counter example)

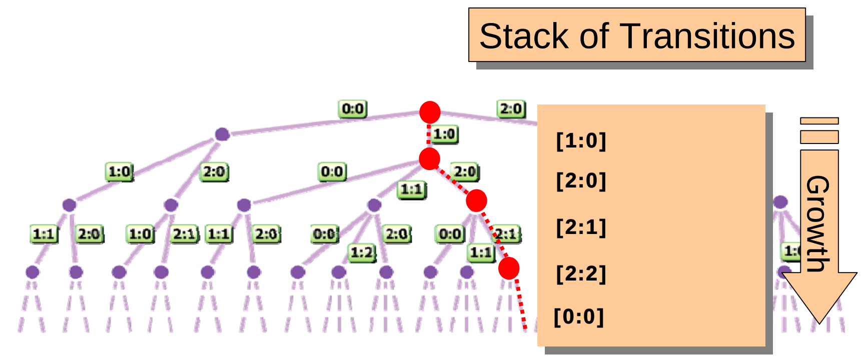
Depth-First Stack (Cont'd)



The depth-first stack can be implemented to hold

- state vectors (straight-forward implementation)

Depth-First Stack (Cont'd)



The depth-first stack can be implemented to hold

- transitions (requires less space, but harder to implement)

Depth-first Stack of Transitions

- Generating a new state s_{new} requires analyzer to execute a transition t on current state s:

  s_{new} = execute(t, s)

- Since analyzer is not holding states in the stack,
  - if it needs to back-track and return to a previously encountered state s_{prev}, it needs to be able to "undo" a transition t

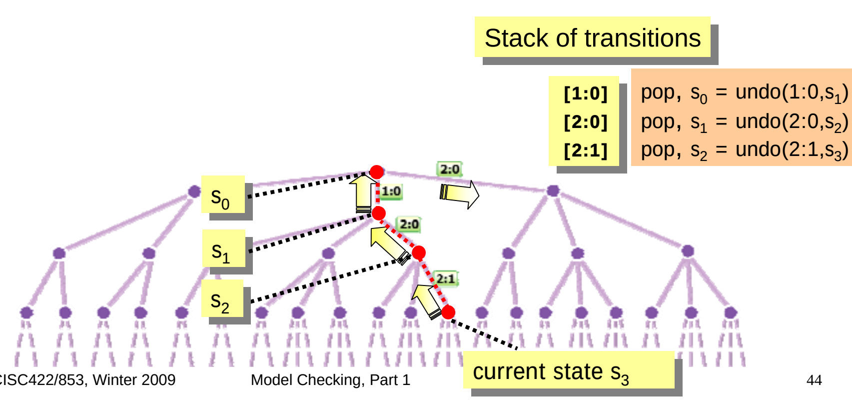
    s_{prev} = undo(t, s)

  - when providing variable values as diagnostic information for an error path, the analyzer needs a "simulation mode" where choice points are decided by the stacked transitions

Depth-first Stack of Transitions (Cont'd)

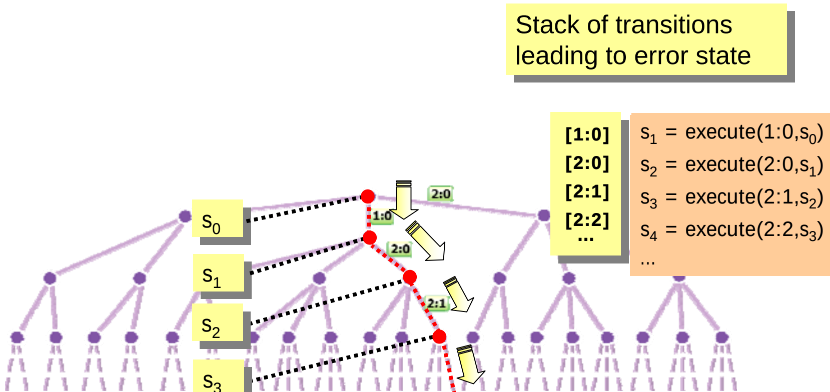
- Since analyzer is not holding states in the stack,
  - if it needs to back-track and return to a previously encountered state s_{prev}, it needs to be able "undo" a transition t

    s_{prev} = undo(t, s)



Depth-first Stack of Transitions (Cont'd)

- Since analyzer is not holding states in the stack,
  - when providing variable values as diagnostic information for an error path, analyzer needs a simulation mode where choice points are decided by the stacked transitions



Source: 842@KSU

Depth-First Stack of Transitions (Cont'd)

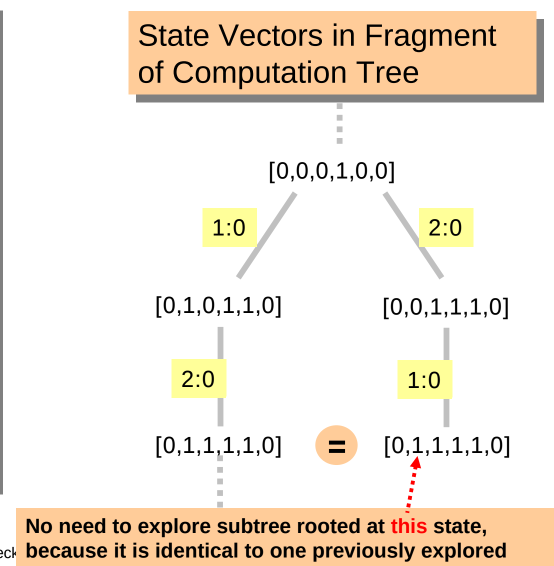
- Many model-checkers (including SPIN and Bogor) implement a depth-first stack of transitions
- This reduces amount of required memory and meshes well with its other space optimizations (e.g., bit-state hashing – discussed in Topic 8)

Seen State Set

- There may be more than one path to a given state
- If a state is reached for a second time, there is no need to check s again (or any of the children of s in the computation tree)
- Seen State Set:
  - used to avoid exploring/checking a part of the computation tree that is identical to a part that has already been explored before
  - in Bogor: implemented as hash table

Revisiting Via A Different Path

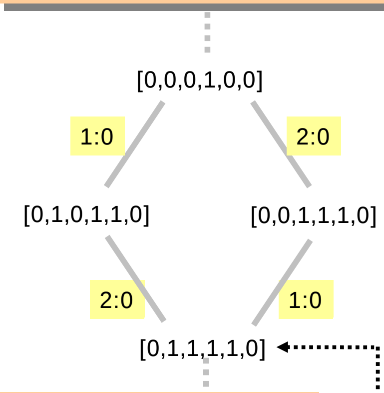
```
      active thread Threadk() {
k:0     loc loc0:
          when x != (byte)0 do {
          t1 := x; } goto loc1;

k:1     loc loc1:
          do { t2 := x; }
          goto loc2;

k:2     loc loc2:
          do { x := t1 + t2; }
          goto loc0;
      }

      active thread Thread0() {
0:0     loc loc0:
          do {
            assert (x !=
              (byte)PARAM.N); }
          return;
        }
      }
```



Computation Tree as Graph

```
k:0  active thread Threadk() {
       loc loc0:
         when x != (byte)0 do {
         t1 := x; } goto loc1;

k:1    loc loc1:
         do { t2 := x; }
         goto loc2;

k:2    loc loc2:
         do { x := t1 + t2; }
         goto loc0;
     }

0:0  active thread Thread0() {
       loc loc0:
       do {
         assert (x !=
           (byte)PARAM.N); }
       return;
     }
}
```

Sometimes we view the computation tree as a graph

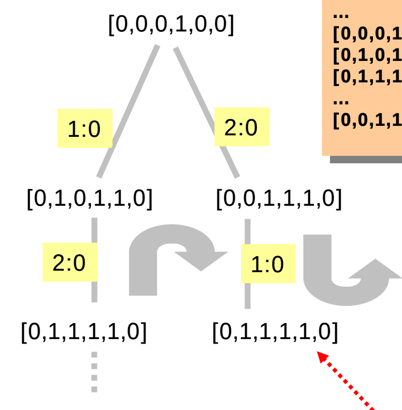


...sharing a node corresponds to (re)visiting a node that has been seen before.

Seen State Set

```
k:0  active thread Threadk() {
       loc loc0:
         when x != (byte)0 do {
         t1 := x; } goto loc1;

k:1    loc loc1:
         do { t2 := x; }
         goto loc2;

k:2    loc loc2:
         do { x := t1 + t2; }
         goto loc0;
     }

0:0  active thread Thread0() {
       loc loc0:
       do {
         assert (x !=
           (byte)PARAM.N); }
       return;
     }
}
```

Computation Tree



Seen Set

...
[0,0,0,1,0,0]
[0,1,0,1,1,0]
[0,1,1,1,1,0]
...
[0,0,1,1,1,0]

When Bogor gets to this state, it checks the Seen Set and finds it already has been checked, so it backtracks from this point

Non-Terminating Systems

- Let S be a finite state machine
  - Due to the use of the Seen Set, checking S will always eventually terminate
  - Even if S has non-terminating executions (Why?)
- Example: Consider the system on the right...
  1. Does execution of the system terminate?
  2. How many states does it have?
  3. Does an exhaustive analysis of the state-space of the system terminate?

```
system Loops

  boolean x;

  active thread Thread1() {
    loc loc0: do { x := !x; }
    goto loc0;
  }

  active thread Thread2() {
    loc loc0: do { x := !x; }
    goto loc0;
  }
}
```

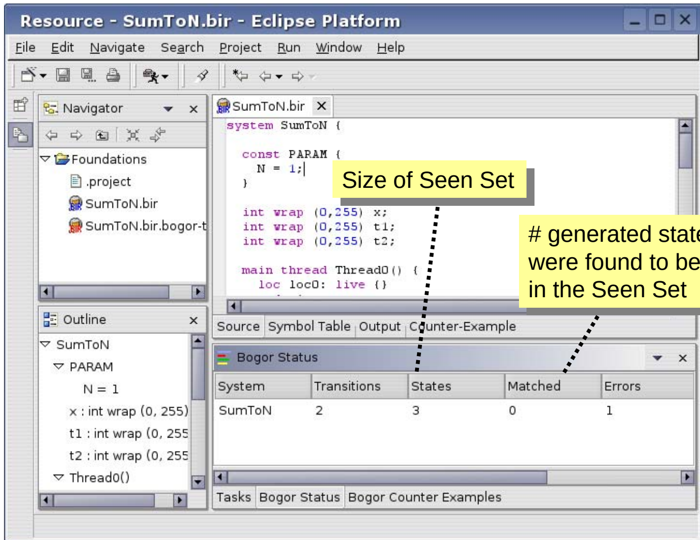

Finite is not Enough

- So, the analysis of every BIR or PROMELA program will always terminate...
- ... but it may take a really long time to do so
- So, state spaces should not only be finite, but also “small enough” for the exploration to be feasible
- State Explosion Problem: Size of state space grows exponentially with the number of parallel processes
- Beware of systems with
  - large numbers of parallel processes
  - variables ranging over large domains (e.g., int, long)

    } too many states; analysis takes too much time
  - variables ranging over large, complex data
  - large numbers of variables

    } states too large; analysis requires too much space

Bogor Output



Bogor Output (Cont'd)

Running a model-check of SumToN with N = 5:

Depth in computation tree (i.e., transition stack) where assertion violation was found (i.e., number of steps in error trace)

[Time: 4817 ms, Depth: 395] Error found: Assertion failed

Transitions: 38174, States: 15276, Matched States: 22899,
Max Depth: 1921, Errors found: 19
Total memory before search: 329,240 bytes (0.31 Mb)
Total memory after search: 4,327,968 bytes (4.13 Mb)
Total search time: 4897 ms (0:0:4)
States count: 15276
Matched states count: 22899
Max depth: 1921

Deepest stack depth reached during search

Checking for Assertion Violations

```
checkAssertions(A_S) {
  for all s0∈ S0 {
    seen := {}
    stack := [s0]
    DFS(s0)
}

DFS(s) {
  ws := enabled(s)
  for all a in ws {
    if a=assert(p) && !eval(p,s) then
      print("violation", s+stack)
    s' := execute(a, s)
    if s' not in seen {
      seen := seen + {s'}
      push(s', stack)
      DFS(s')
      pop(stack)
}}}
```

- seen := {} — **set of states already explored**
- stack := [s0] — **states on current path**
- ws := enabled(s) — **get the transitions out of s (possibly "on-the-fly")**
- for all a in ws — **pick one of the transitions to explore**
- if a=assert(p) && !eval(p,s) then — **check for assertion violation, if necessary**
- s' := execute(a, s) — **calculate the successor state**
- if s' not in seen — **if s has been seen before, ignore it**
- DFS(s') — **explore successor state**

How does the algorithm have to be modified to check for deadlock?

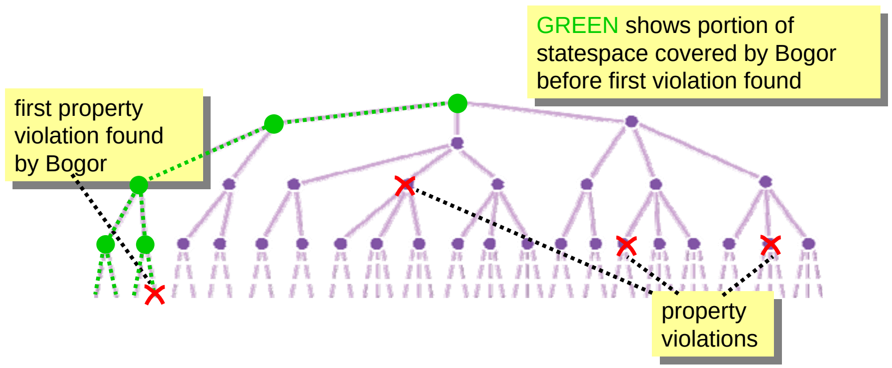
Source: 842@KSU

Error Trace Length

- Model-check SumToN with N = 5
- From Bogor's output, can see that execution trace that violates assertion was found and that trace is 395 steps long
  - Having to reason about how assertion can be violated along a trace of 395 steps is quite painful!
  - You have previously discovered a much shorter violating trace using Bogor's simulation mode.
  - Does this mean that the Bogor analyzer is not very useful?
    - Not at all!!
- We will see now how to tell Bogor to search for shorter violating traces (as well as minimal length violating traces)

Error Trace Length

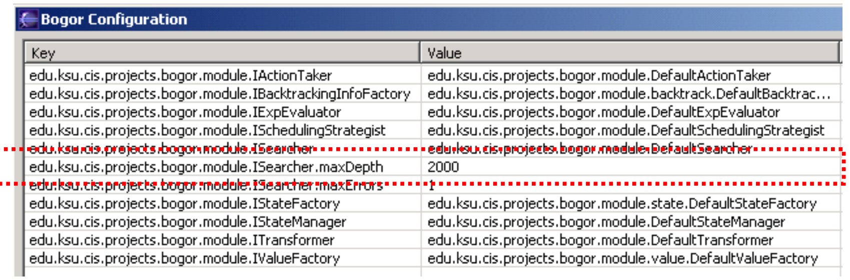
- In general, a system may have many different traces that lead to the same property violation



- Because Bogor does a depth-first search (instead of a bread-first search), first violating trace found is usually not of minimal length

Setting Bogor's Depth Bound

- Users can set a bound on the depth of Bogor's search (i.e., number of entries in Bogor's depth-first stack)



Bogor Configuration

Key	Value
edu.ksu.cis.projects.bogor.module.IActionTaker	edu.ksu.cis.projects.bogor.module.DefaultActionTaker
edu.ksu.cis.projects.bogor.module.IBacktrackingInfoFactory	edu.ksu.cis.projects.bogor.module.backtrack.DefaultBacktrac...
edu.ksu.cis.projects.bogor.module.IExpEvaluator	edu.ksu.cis.projects.bogor.module.DefaultExpEvaluator
edu.ksu.cis.projects.bogor.module.ISchedulingStrategist	edu.ksu.cis.projects.bogor.module.DefaultSchedulingStrategist
edu.ksu.cis.projects.bogor.module.ISearcher	edu.ksu.cis.projects.bogor.module.DefaultSearcher
edu.ksu.cis.projects.bogor.module.ISearcher.maxDepth	2000
edu.ksu.cis.projects.bogor.module.ISearcher.maxErrors	1
edu.ksu.cis.projects.bogor.module.IStateFactory	edu.ksu.cis.projects.bogor.module.state.DefaultStateFactory
edu.ksu.cis.projects.bogor.module.IStateManager	edu.ksu.cis.projects.bogor.module.DefaultStateManager
edu.ksu.cis.projects.bogor.module.ITransformer	edu.ksu.cis.projects.bogor.module.DefaultTransformer
edu.ksu.cis.projects.bogor.module.IValueFactory	edu.ksu.cis.projects.bogor.module.value.DefaultValueFactory

Choose the "Configure Bogor" option, then Add/Edit to set the value for the ISearcher.maxDepth attribute.

Setting Bogor's Depth Bound

- This is often useful…
  - …after a counterexample has been found and you want to see if a shorter one exists
    - look at Bogor's output to see the size, then rerun Bogor with an appropriate depth bound (i.e., one smaller than the size of the counter-example).
  - …before a counterexample has been found and Bogor is taking too long or is running out of memory

Setting Bogor's Depth Bound (Cont'd)

- Be careful!
  - when search is bounded, Bogor will not explore parts of state space
  - unexplored part may contain property violations
  - If a bounded search does not find any violations, then
    - no violations in parts that got searched
    - but may have violations in unsearched parts

  ⇒ A depth-bounded search may be incomplete!
- Bogor displays "Max depth reached!!!" whenever depth bound is reached and analysis may be incomplete

For Example

Checking SumToN with N = 5

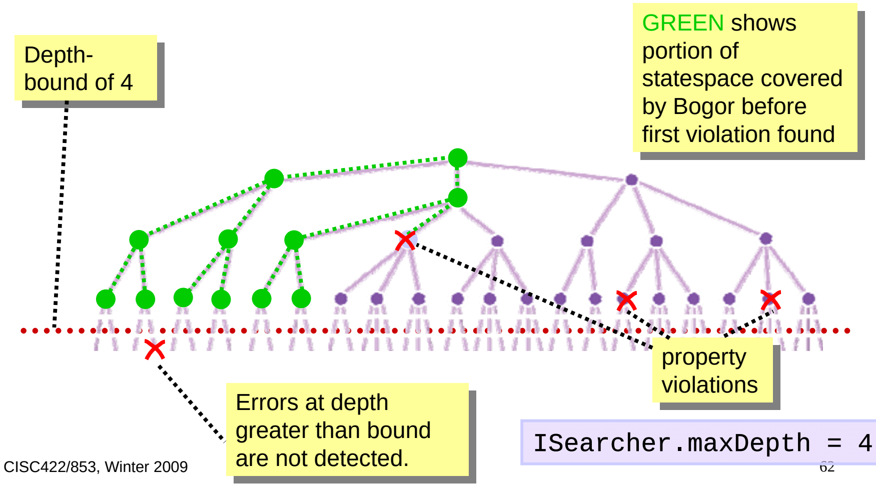
Bogor Status

System	Transitions	States	Matched	Max. Depth	Errors	Time	Status
SumToN	38174	15276	22899	1921	19	0:0:4	Done
SumToN	25	26	0	25	1	0:0:0	Done
SumToN	25	26	0	25	1	0:0:0	Done
SumToN	123	63	61	6	0	0:0:0	Done
SumToN	123	63	61	6	0	0:0:0	Done
SumToN	39	23	17	3	0	0:0:0	Done
SumToN	39	23	17	3	0	0:0:0	Done
SumToN	463	225	239	12	1	0:0:0	Done
SumToN	382	189	194	11	1	0:0:0	Done
SumToN	296	155	142	10	1	0:0:0	Done
SumToN	275	134	142	9	0	0:0:0	Done

Tasks | Problems | Bogor Status

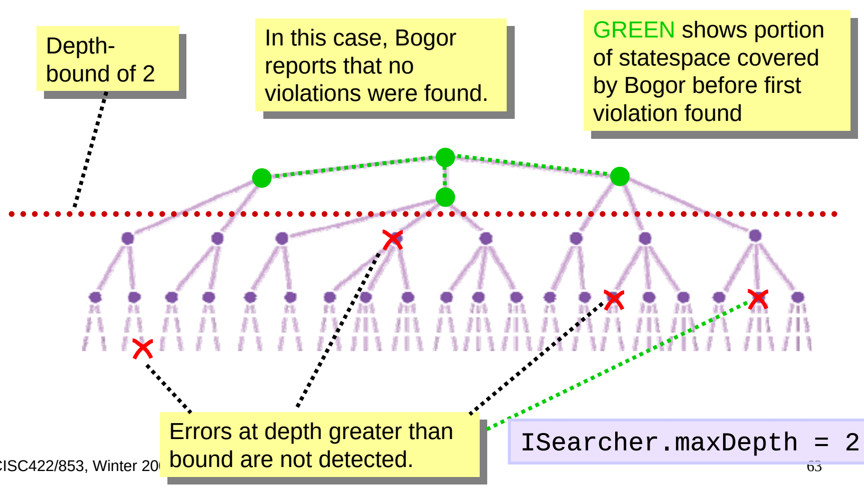
Bounded Depth-first Search

When analyzing a system and given a depth bound as a command-line argument, Bogor will backtrack when that depth is reached



Bounded Depth-first Search (Cont'd)

When analyzing a system and given a depth bound as a command-line argument, Bogor will backtrack when that depth is reached



Depth-Bounded DFS

- Advantages: ?
- Disadvantages: ?

Finding the Shortest Counter Example

- Using Bounded DFS
  - in Bogor:
    - start with high bound that finds error
    - successively lower the bound until no error
  - in Spin:
    - Run verifier with option –i or –l:

      `pan.exe -i` or `pan.exe -l`
- Using ?

Yes, Breadth First Search!

- How to make Bogor and Spin use BFS
  - in Bogor:
    - write routine and plug it in
    - modular architecture of Bogor makes this easy
  - in Spin:
    - compile verifier with –DBFS option:

      `gcc -DBFS -o pan pan.c`
- Easy to implement
- What’re the advantages of BFS over DFS?
- What’re the disadvantages?