Domain-specific Model Checking with Bogor

SAnToS Laboratory, Kansas State University, USA

http://bogor.projects.cis.ksu.edu

Robby   Matthew B. Dwyer   John Hatcliff

Matthew Hoosier

Session II: DFS in Explicit Model Checking

Support

US Army Research Office (ARO)
US National Science Foundation (NSF)
US Department of Defense
Advanced Research Projects Agency (DARPA)

Boeing
Honeywell Technology Center
IBM
Intel

Lockheed Martin
NASA Langley
Rockwell-Collins ATC
Sun Microsystems

SumToN

declare a namespace PARAM with a constant N so that we can easily modify N's value.

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

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

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

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

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

```
      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;
  }
}
```

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:
      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:
      do { t1 := x; }
      goto loc1;

      goto loc0;
  }

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

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

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:
      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:
      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;
  }
}
```

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:
      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:
      do { t1 := x; }
      goto loc1;

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

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

  Thread0() {
  != (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.

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:
      do { t1 := x; }
      goto loc1;

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

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

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

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

          1:
          t2 := x; }
          loc2;

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

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

SumToN

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

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

  active threa
    loc loc0:
      do { t1 :=
      goto loc1;

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

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

Note: *This transition can be arbitrarily interleaved with all others from Thread1 and Thread2.*

```
  active thread Thread2() {
    loc loc0:
      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;
  }
}
```

Assessment

Pick a value of N (e.g., 5) Can the assertion in the SumToN example be violated (i.e., can x ever have the value 5)?

- 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() {
k:0   loc loc0:
        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 $N = 1$

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

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

violation

Move this thread first….

…that was easy!

SumToN Assertion Violation

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

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

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

 active thread Thread0() {
  loc loc0:                      0:0
    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]
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

Move only Thread1 until x = 2, then check assertion

SumToN Assertion Violation

```
k:0   active thread Threadk()   Another
        loc loc0:
          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]
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

Move only Thread2 until x = 2, then check assertion

SumToN Assertion Violation

Yet Another

```
    active thread Thread1() {
k:0   loc loc0:
        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 $N = 2$

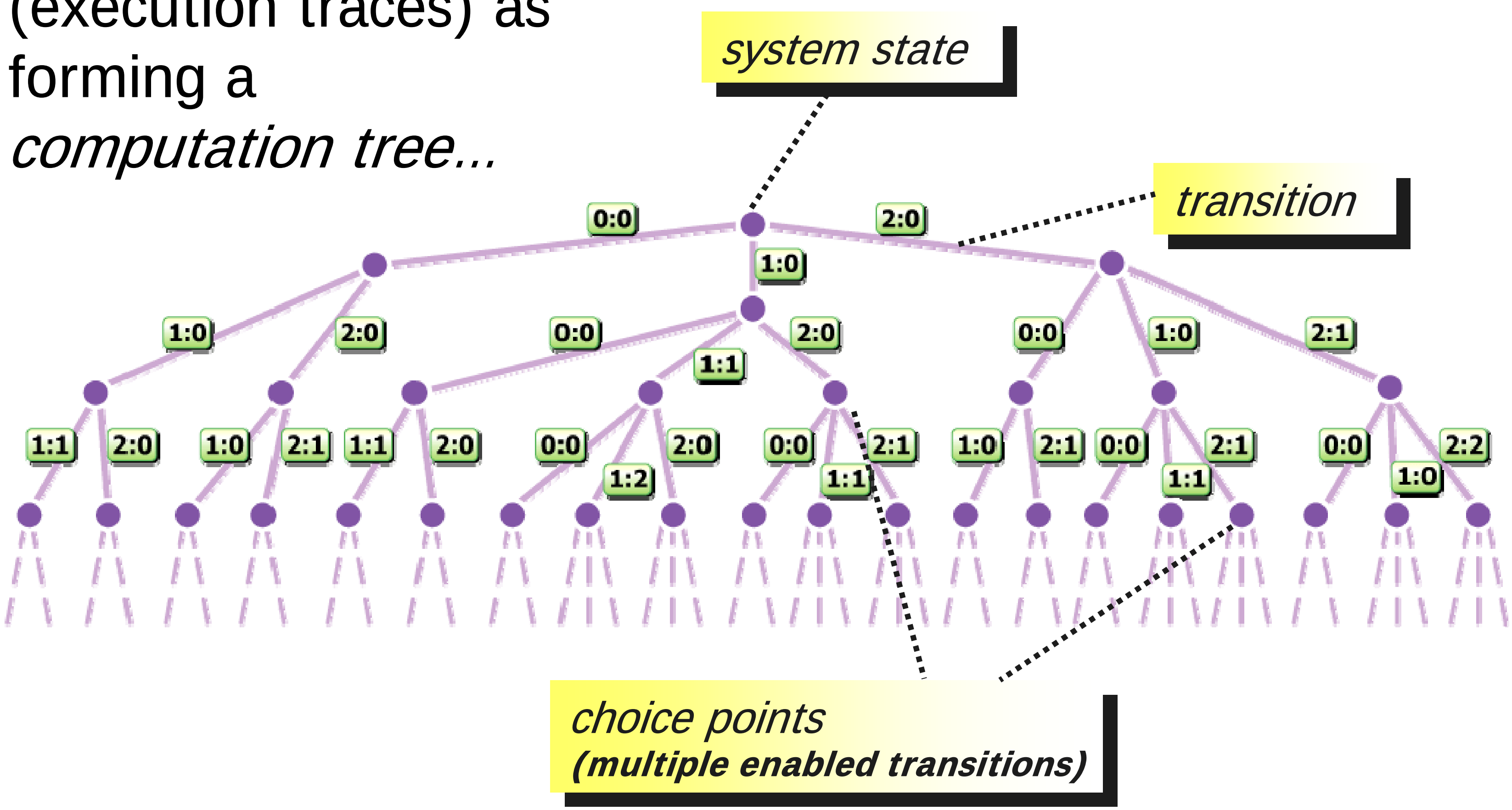
(initial values)	[0, 0, 0, x = 1, t1 = 0, t2 = 0]
1:0	[0, 1, 0, x = 1, t1 = 1, t2 = 0]
2:0	[0, 1, 1, x = 1, t1 = 1, t2 = 0]
2:1	[0, 1, 2, x = 1, t1 = 1, t2 = 1]
2:2	[0, 1, 0, x = 2, t1 = 1, t2 = 1]
0:0	[-, 1, 0, x = 2, t1 = 1, t2 = 1]

violation

Move only Thread1 for one step, then move Thread2 three steps as before....

Computation Tree

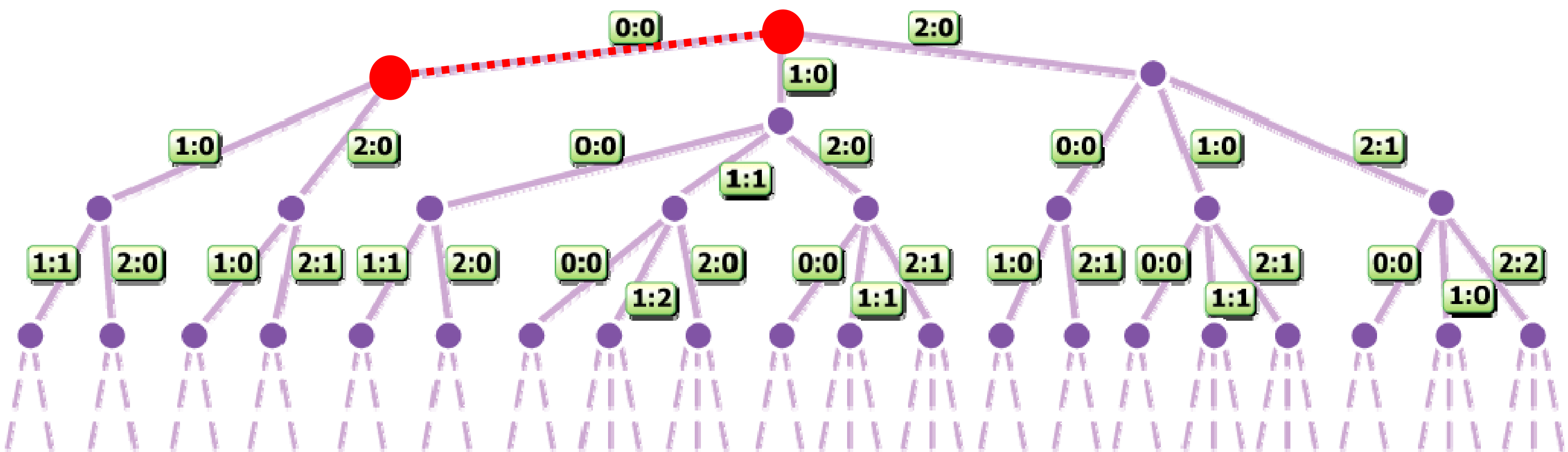
- We can think of the possible schedules (execution traces) as forming a *computation tree*...



Computation Tree

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

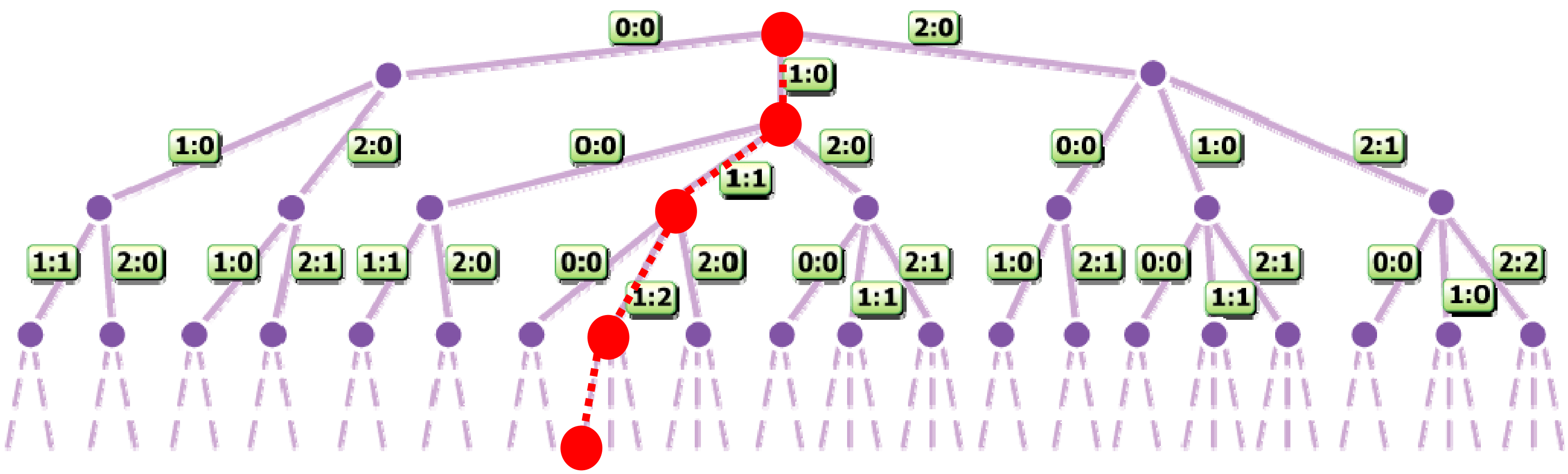
First example trace (schedule)



Computation Tree

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

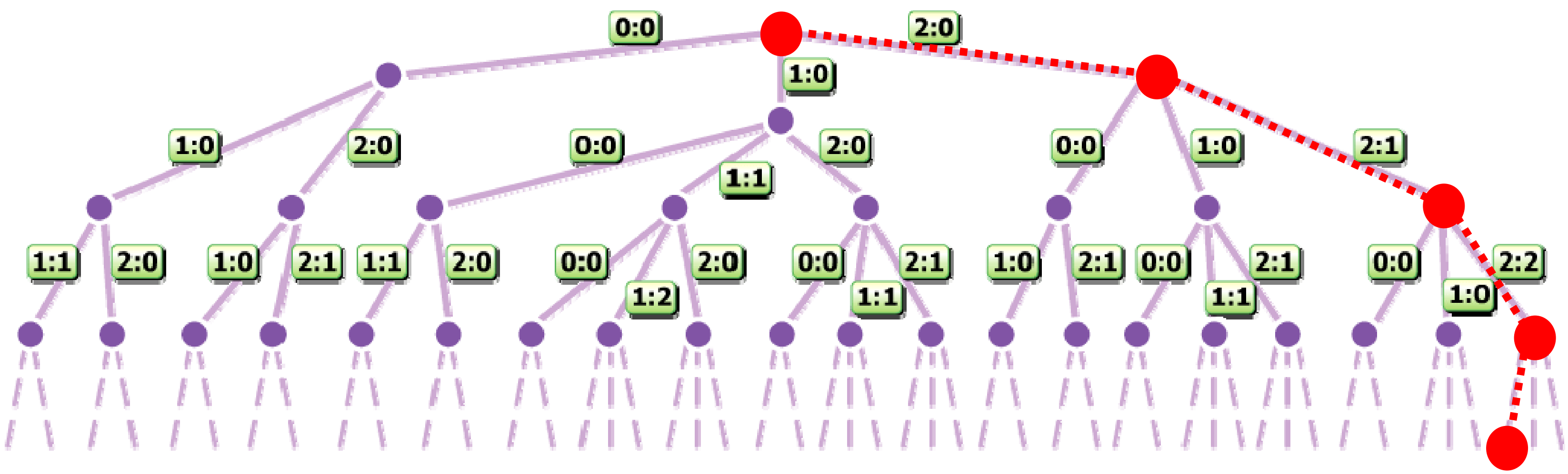
Second example trace (schedule)



Computation Tree

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

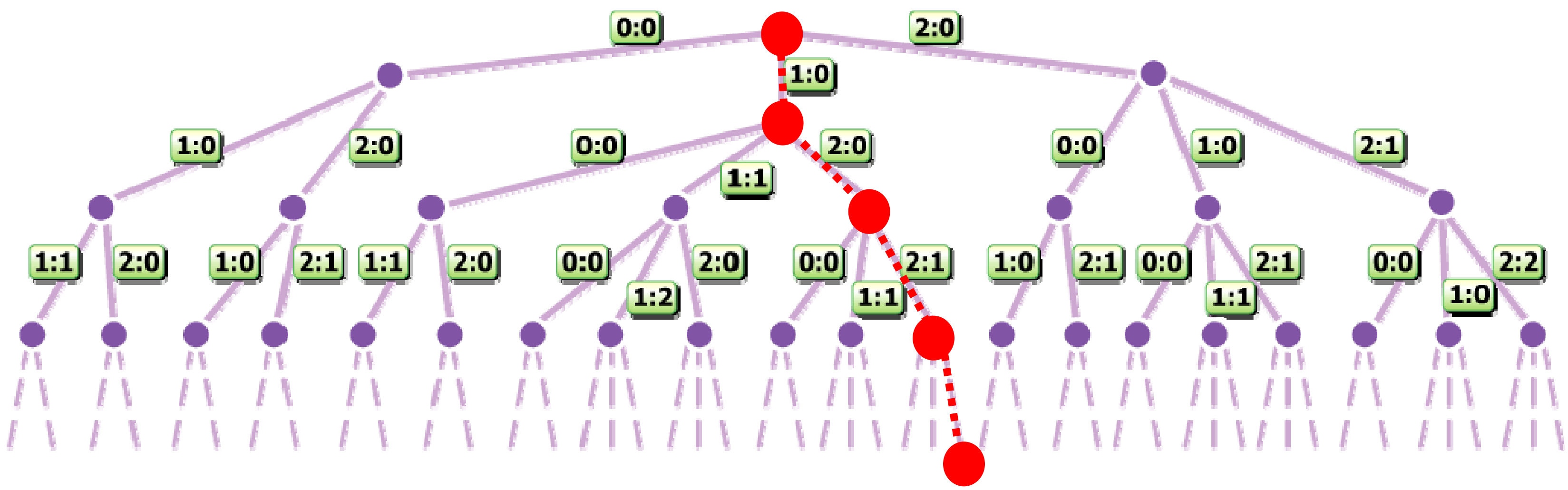
Third example trace (schedule)



Computation Tree

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

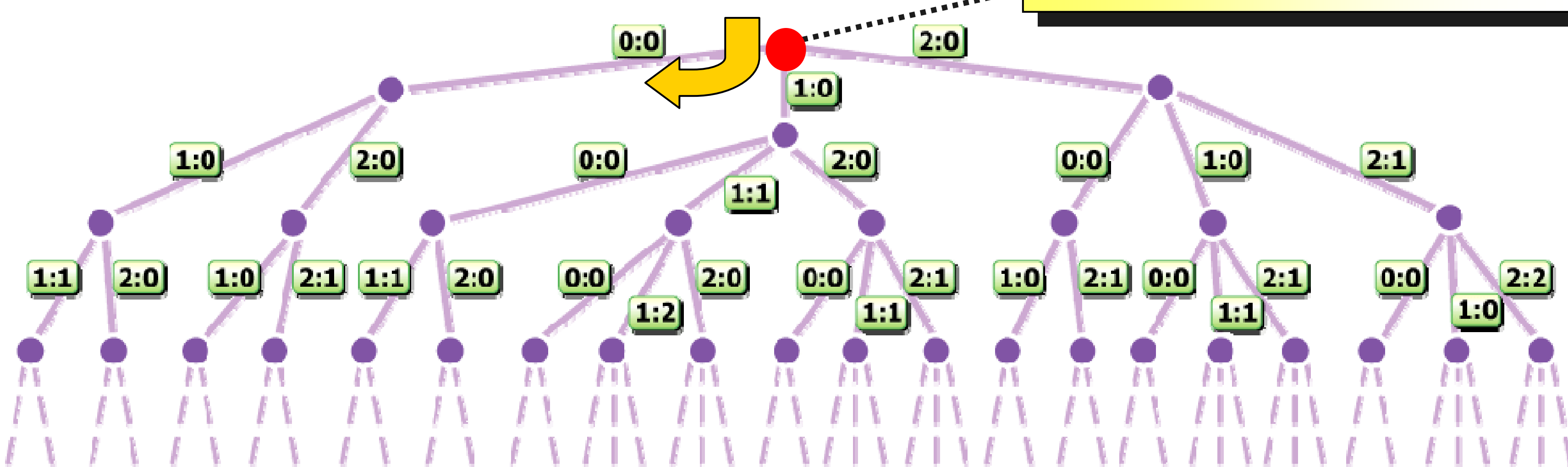
Fourth example trace (schedule)



Exhaustive Depth-first Search

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

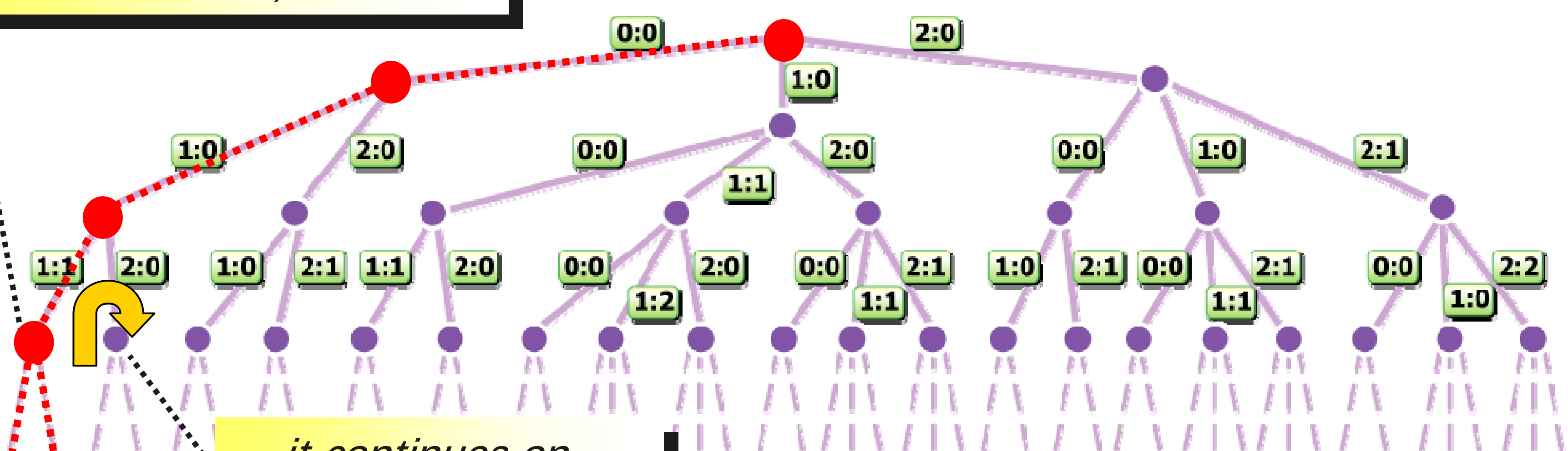
At choice points, Bogor chooses an unexplored transition and remembers that it needs to come back and explore the others…



Exhaustive Depth-first Search

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

When Bogor has finished with one subtree, ...

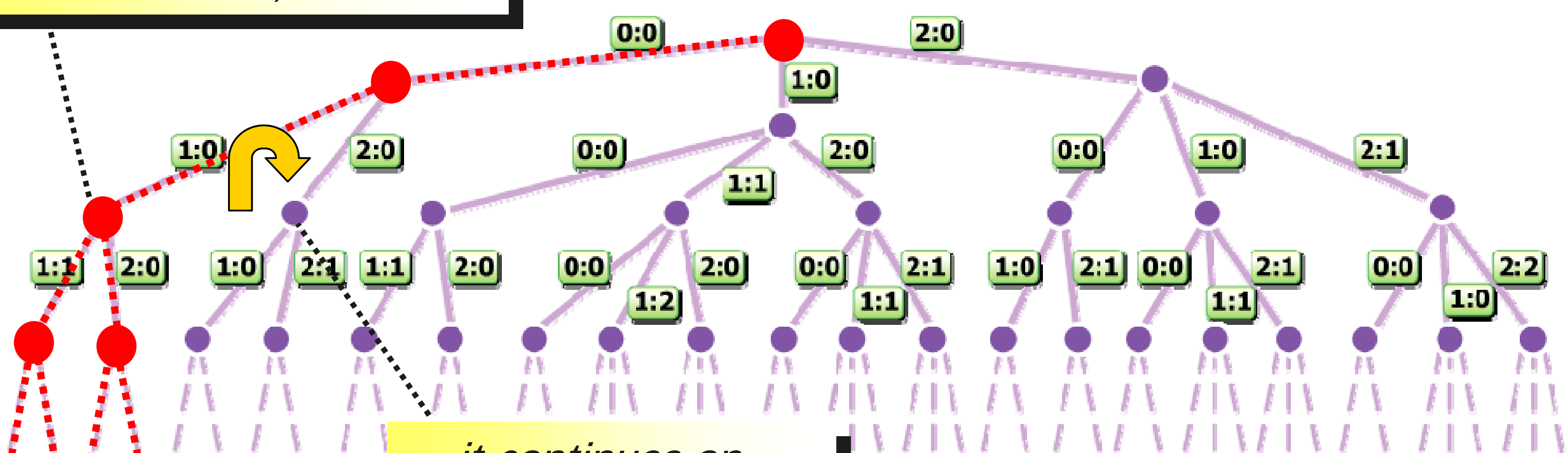


... it continues on with the siblings.

Exhaustive Depth-first Search

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

When Bogor has finished with one subtree, …

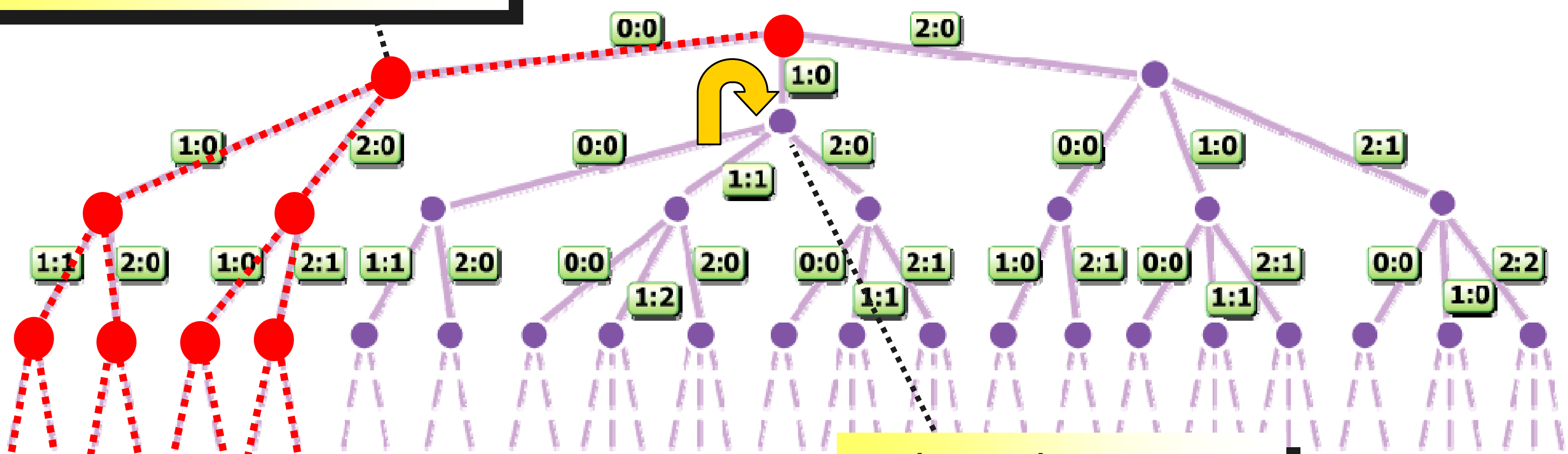


… it continues on with the siblings.

Exhaustive Depth-first Search

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

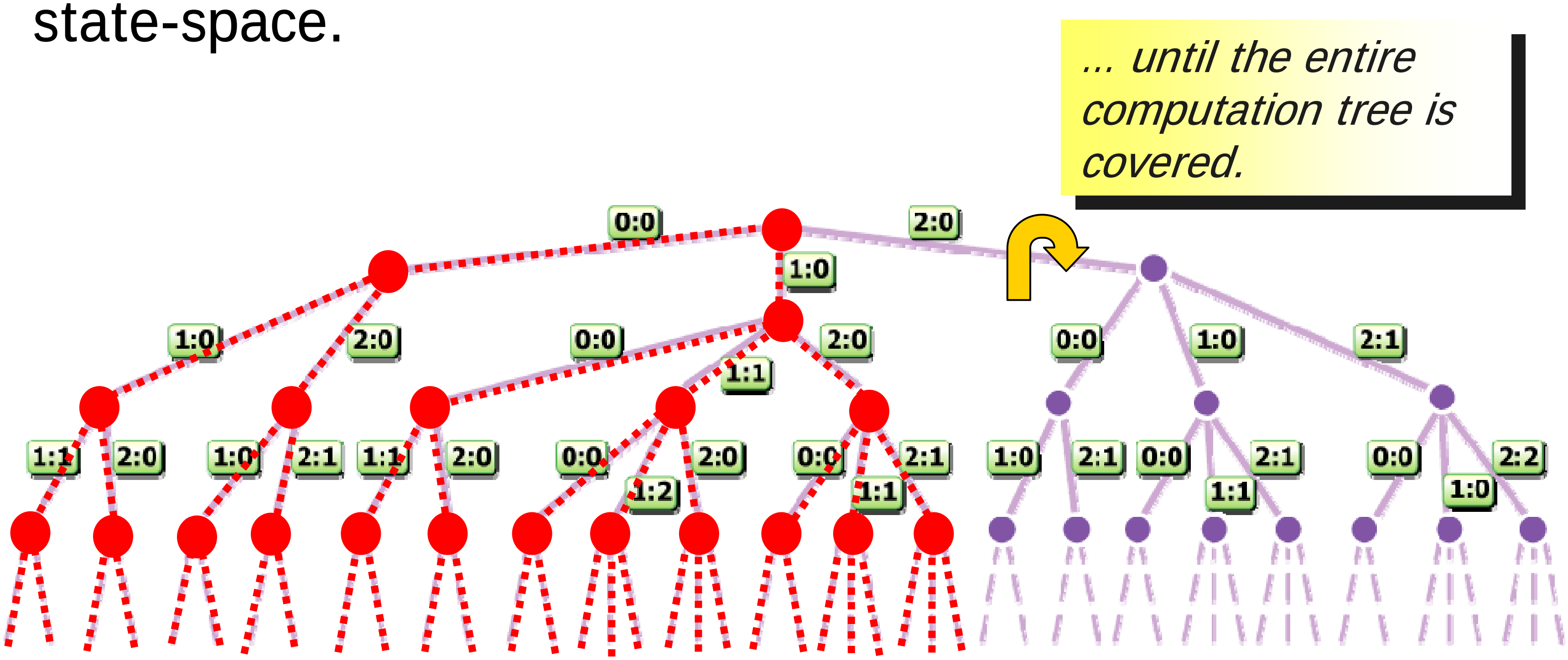
When Bogor has finished with one subtree, …



… it continues on with the siblings.

Exhaustive Depth-first Search

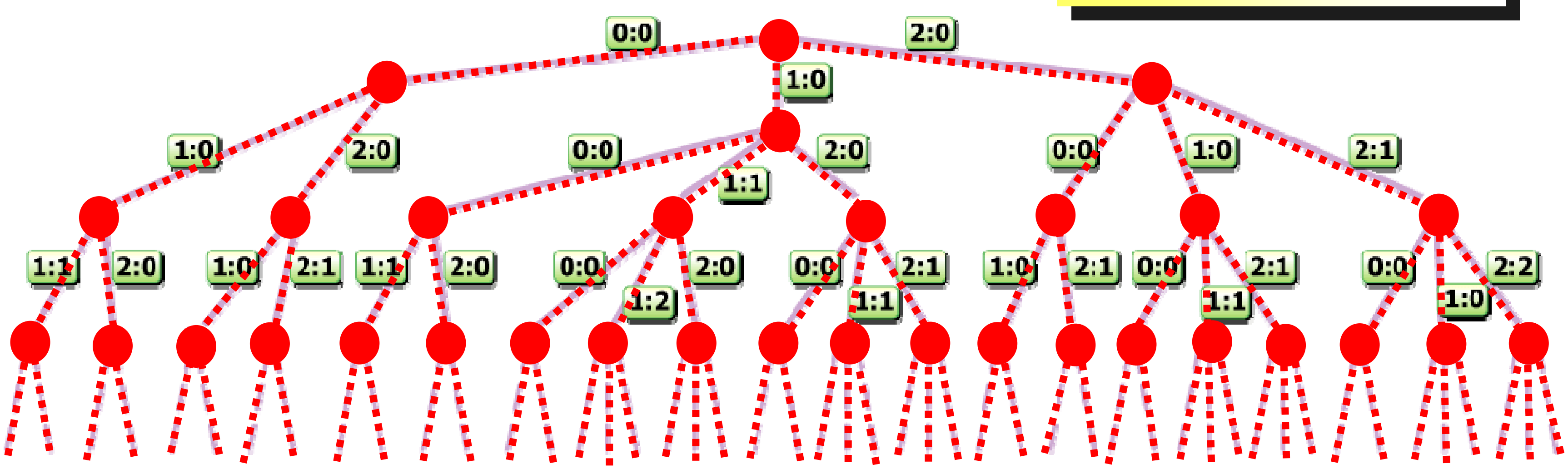
- Bogor can perform exhaustive depth-first searches of a system's state-space.



Exhaustive Depth-first Search

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

... until the entire computation tree is covered.



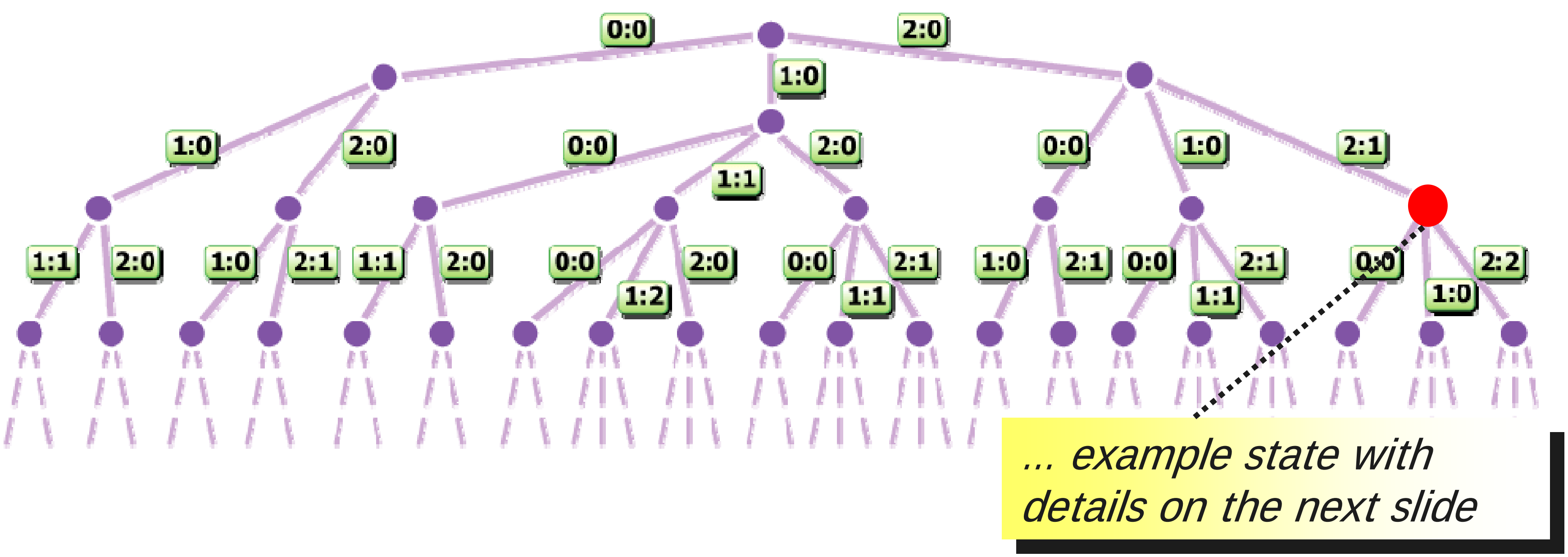
DFS Basic Data Structures

- State vector
  - holds the value of all variables as well as program counters (current position of execution) for each process, and indicates a particular position in the computation tree (as previously covered when discussing state transition systems for BIR).
- Depth-first stack
  - holds the states (or transitions) encountered down a certain path in the computation tree.
- Seen state set
  - holds the state vectors for all the states that have been checked already (seen) in the depth-first search.

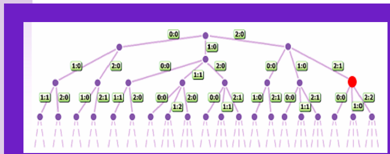
Note: we will represent the values of these data structures in an abstract manner that captures the essence of the issues, but not the actual implementation. Bogor and most other model-checkers actually use multiple clever representations to obtain a highly space/speed optimized search algorithm.

SumToN State Vector Example

- The state vector is the data structure corresponding to the *state* (as previously covered when discussing state transition systems for BIR). It holds the value of all variables as well as program counters for each process, and indicates a particular position in the computation tree.



SumToN State Vector Example



```
{
N = 1 };
int wrap (0,25

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

active thread Thread1() {
  loc loc0:
    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:
    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

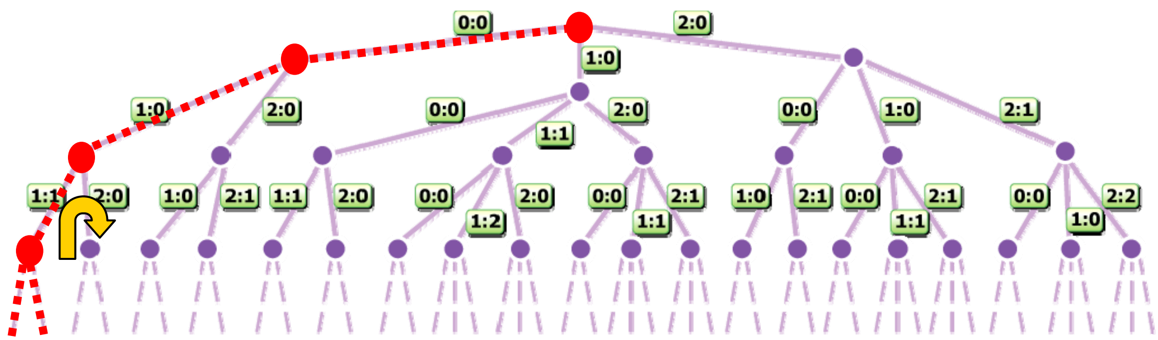
```
active thread Threadk() {
  loc loc0:                          k:0
    do  { t1 := x; }
    goto loc1;

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

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

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

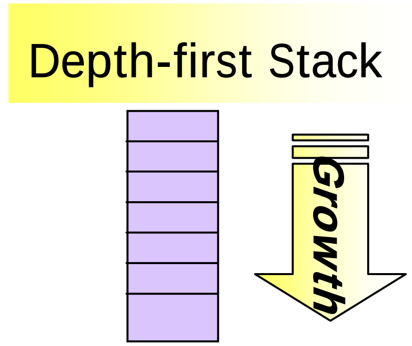
Violating schedule for $N = 2$

(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

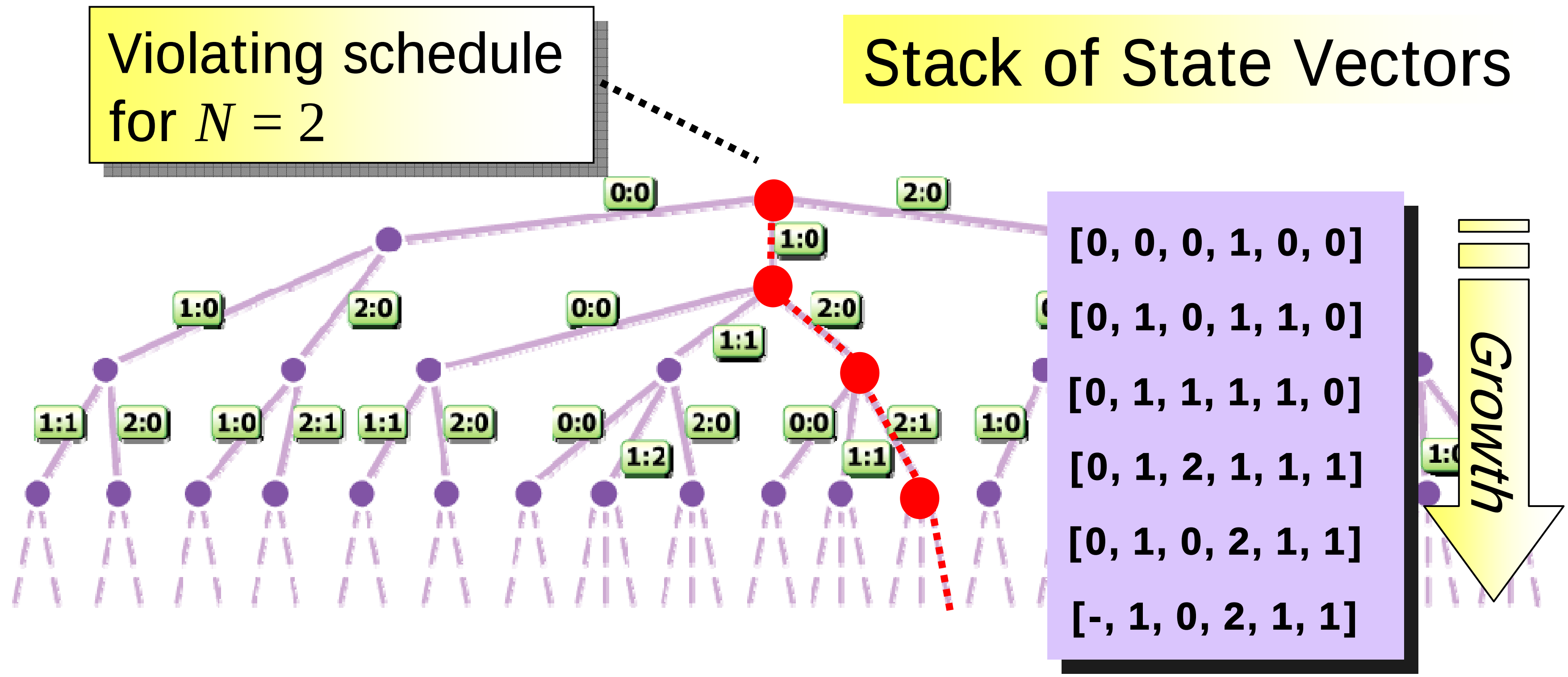
Depth-first Stack





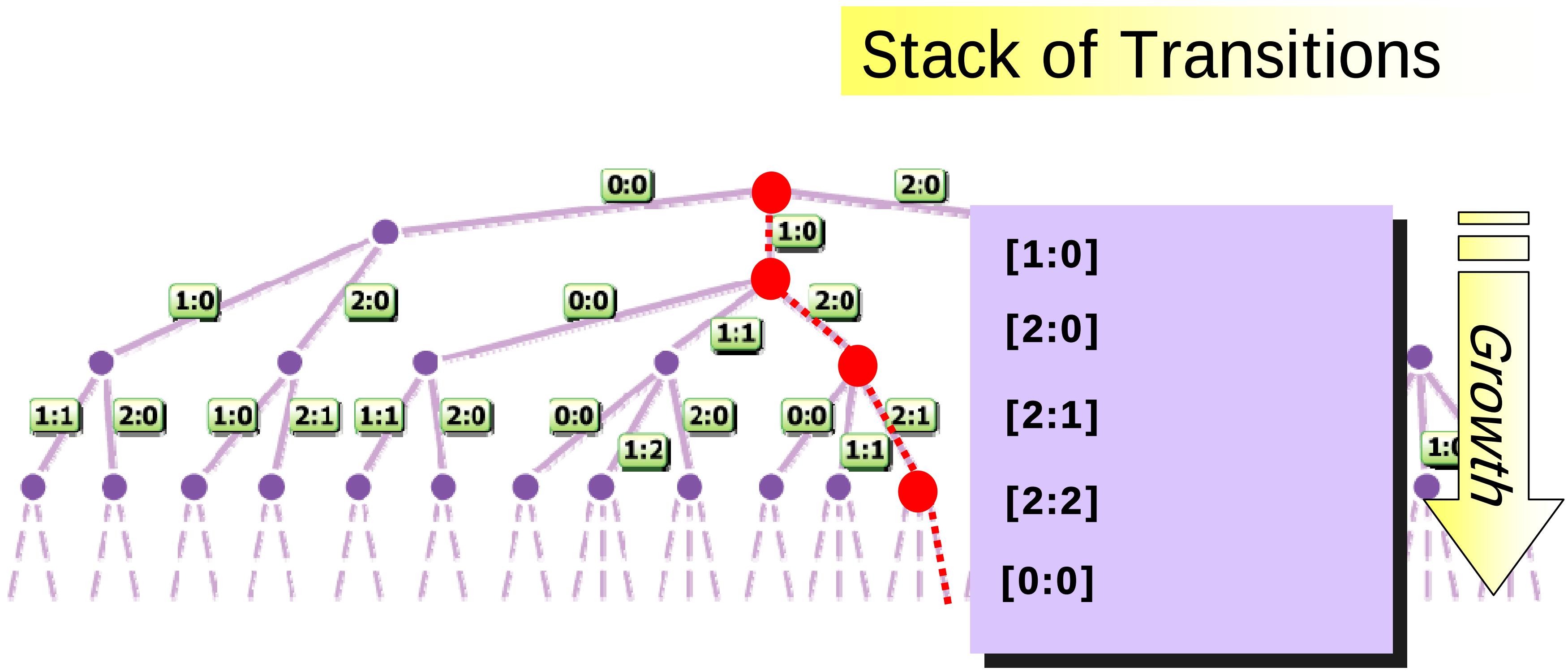
- The depth-first stack serves two purposes
  - When we come to the end of a path (or a state that we have seen before) and backtrack, the stack tells us where to backtrack to.
  - If an error is encountered, the current value of the stack gives the computation path that leads to the error.

Depth-first Stack



- The depth-first stack can be implemented to hold state vectors
  - straight-forward implementation

Depth-first Stack



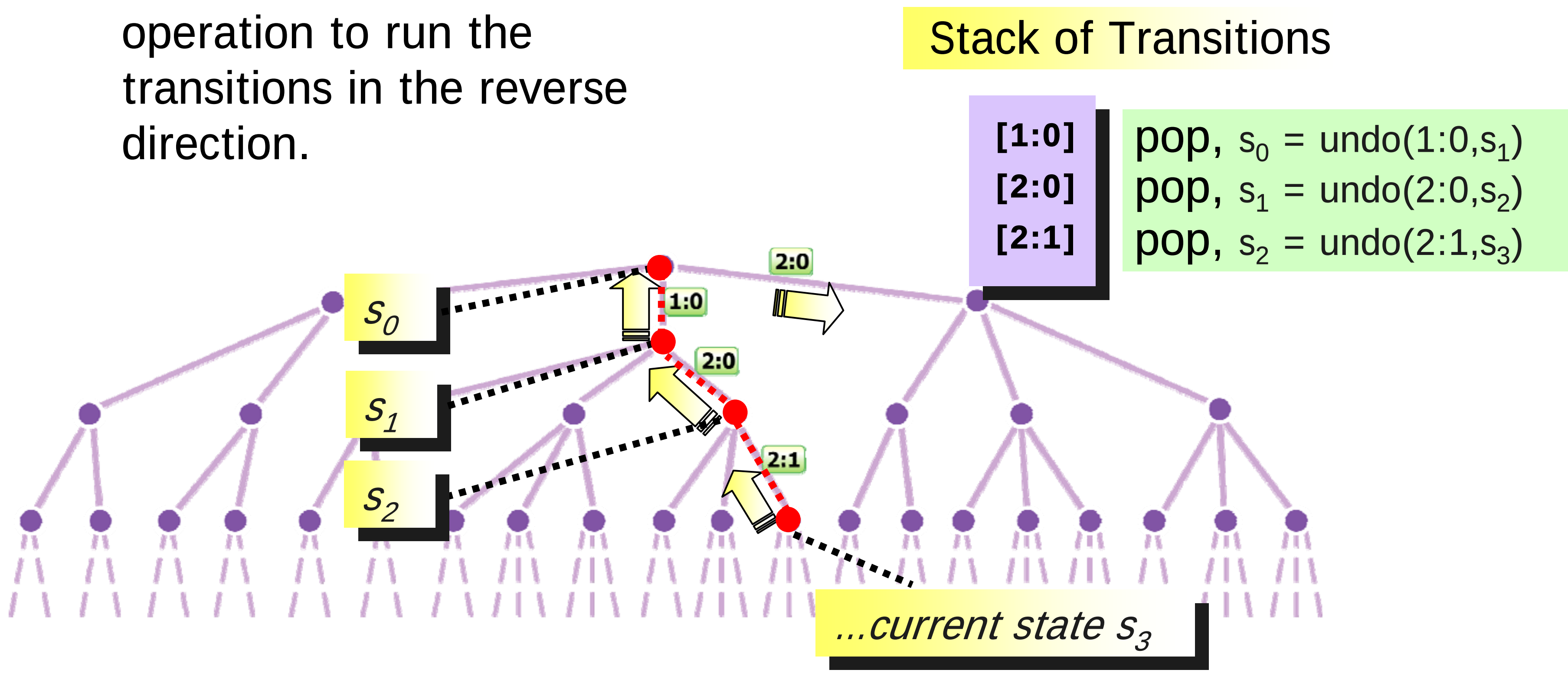
- The depth-first stack can be implemented to hold transitions
  - requires less space, but ...(see next slide)...

Depth-first Stack of Transitions

- Generating a new state requires that the analyzer run a transition on the current state.
- Since the analyzer is not holding states in the stack, if it needs to back-track and return to a previously encountered state, it needs an "undo" operation to run the transitions in the reverse direction.
- Since the analyzer is not holding states in the stack, 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.

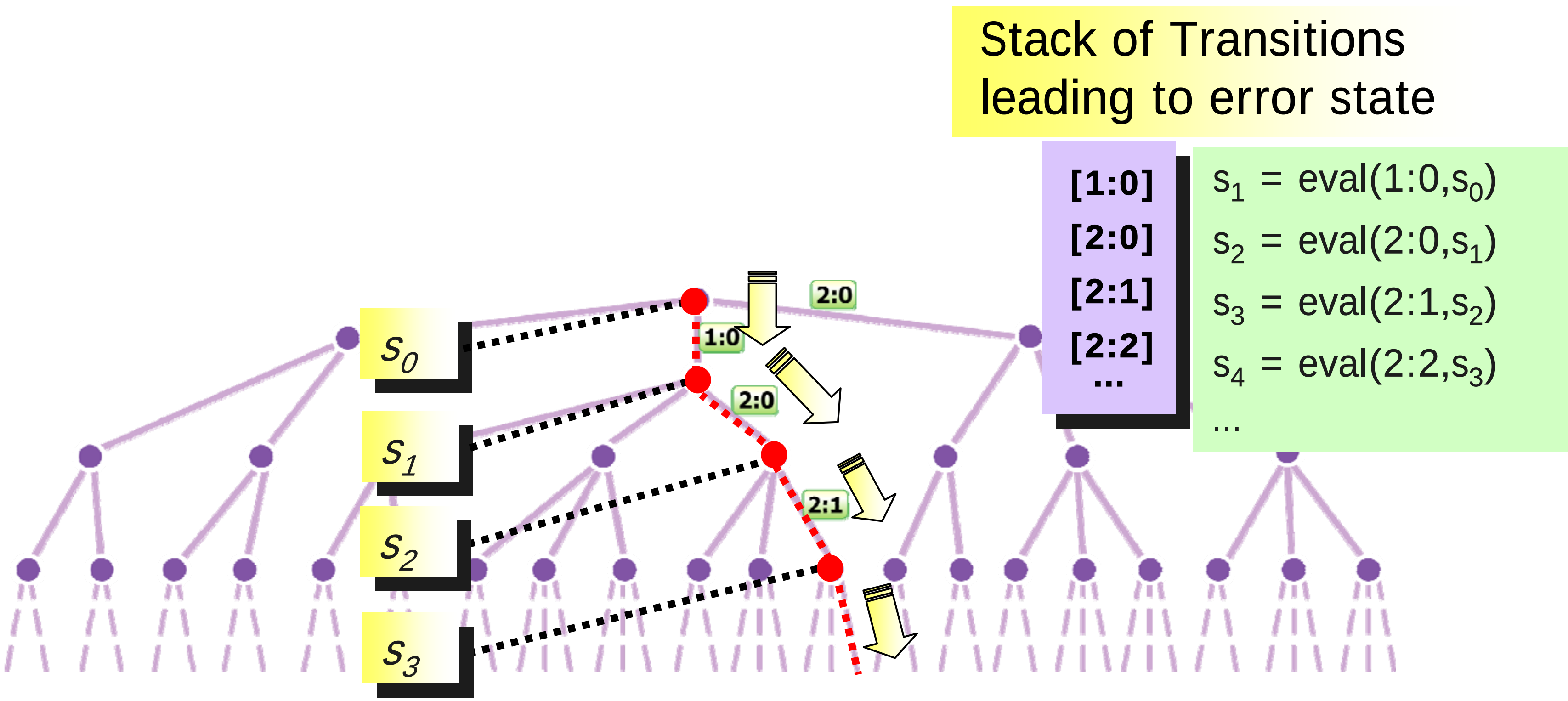
Depth-first Stack of Transitions

- Since the analyzer is not holding states in the stack, if it needs to back-track and return to a previously encountered state, it needs an "undo" operation to run the transitions in the reverse direction.



Depth-first Stack of Transitions

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

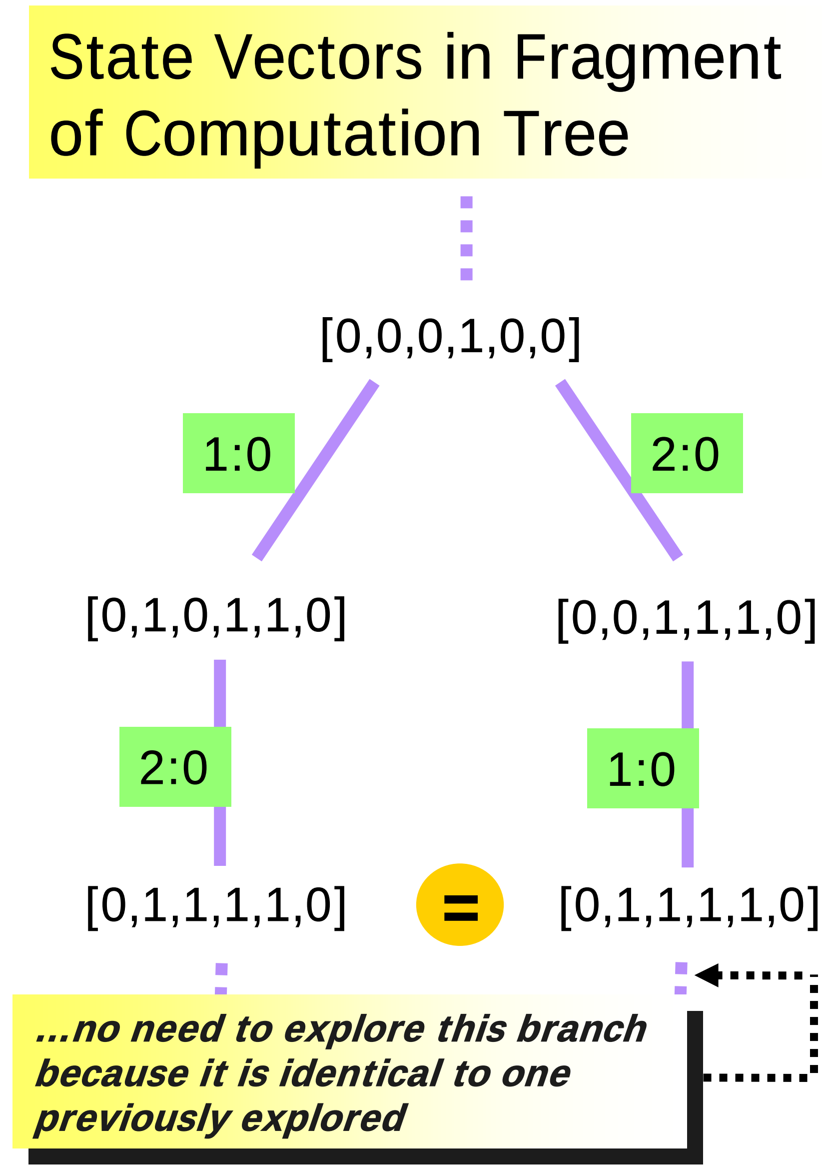


Seen State Set

- Often the analyzer will proceed along a different path to a state S that it has checked before.
- In such a case, there is no need to check S again (or any of S's children in the computation tree) since these have been checked before.
- Bogor maintains a Seen State set (implemented as a hash table) of states that have been seen before, and it consults this set to avoid exploring/checking a part of the computation tree that is identical to a part that has already been explored before.

Revisting Via A Different Path

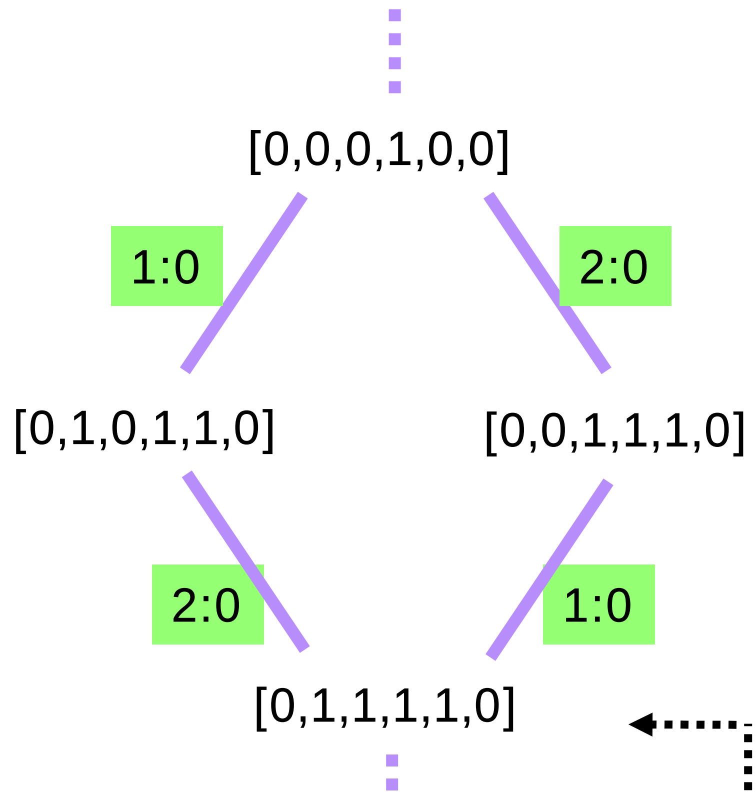
```
active thread Threadk() {
  loc loc0:
k:0  do  { t1 := x; }
     goto loc1;

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

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

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



Computation Tree as Graph

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

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

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

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

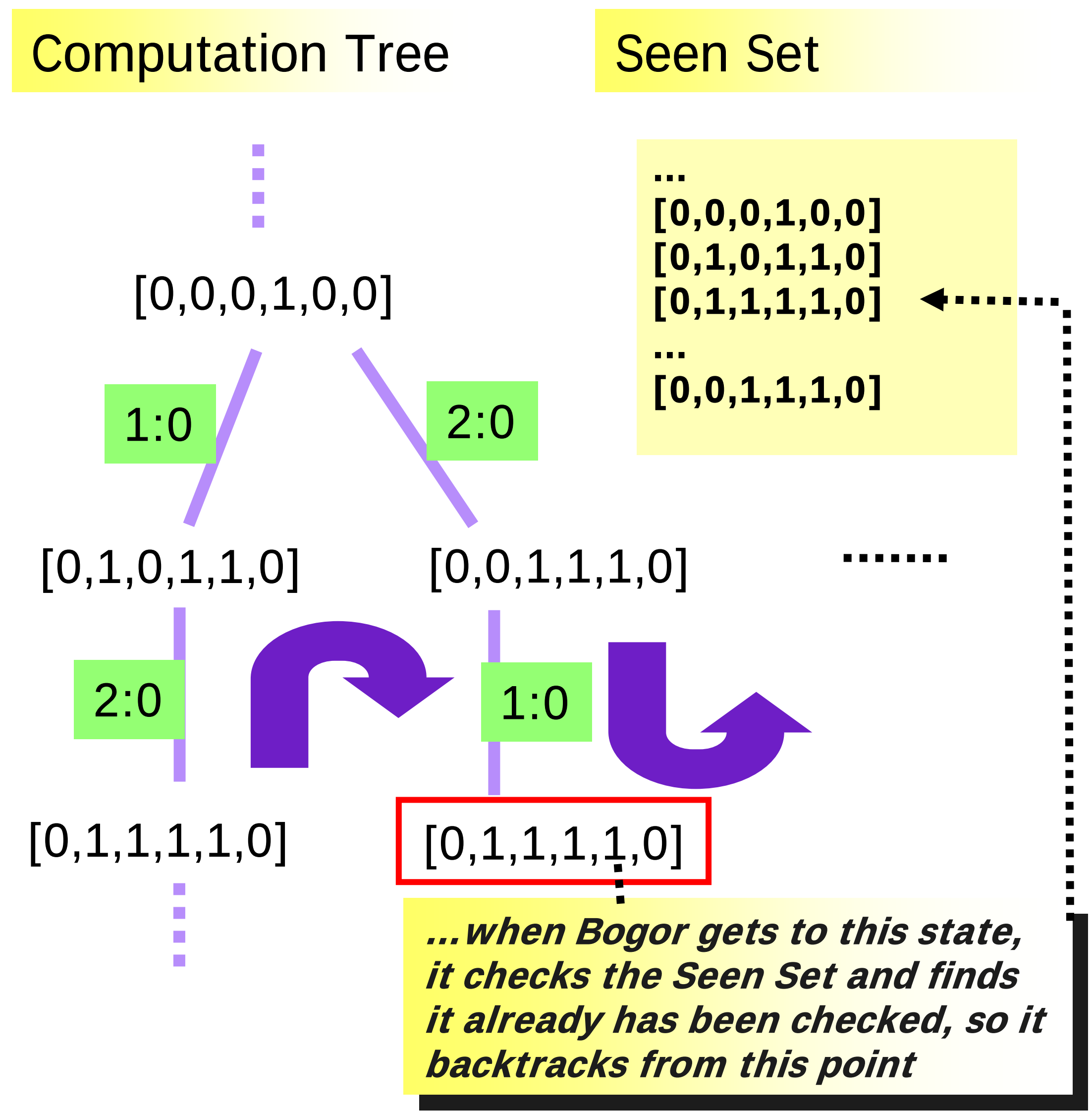
Some times we view the computation tree as a graph



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

Seen State Set

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

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

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

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



Non-Terminating Systems

- Due to the use of the Seen Set, checking a non-terminating system may terminate if the system only has a finite number of states.
- In *basic* BIR, all systems are “finite” because of the bounds on basic data types.
- However, some systems are “more finite” than others.
  - i.e., they have a much smaller state-space.

Non-Terminating Systems

```
system Loops {

  boolean x;

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

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

- Consider this example system...
  - How many states does it have?
  - Does execution of the system terminate?
  - Does an exhaustive analysis of the state-space of the system terminate?