

# CIS 842: Specification and Verification of Reactive Systems

## Lecture INTRO-Computation-Trees: Interleavings, Schedules, and Computation Trees

Copyright 2004, Matt Dwyer, John Hatcliff, and Robby. The syllabus and all lectures for this course are copyrighted materials and may not be used in other course settings outside of Kansas State University in their current form or modified form without the express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.

## Objectives

- Understand the concept of thread interleaving in concurrent systems
- Understand how a system's schedules can be viewed as a *computation tree* where a path through the tree corresponds to a particular schedule
- Be able to draw the computation tree for any simple BIR-Lite system

# Outline

- The SumToN Example
  - use this simple BIR-Lite system to explain the concepts of schedule and computation tree
- Computation Trees

# SumToN

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

  byte x;
  byte t1;
  byte t2;

  thread Thread10 {
    loc loc0:
      when x != 0 do { t1 := x; }
      goto loc1;

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

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

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

```
  when x != 0 do { t1 := x; }
  goto loc1;

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

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

  thread Thread00 {
    loc loc0:
      do {
        x := 1;
      } goto loc1;

    loc loc1:
      do { assert (x != PARAM.N); }
      return;
  }
}
```

# SumToN

```

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

  byte x;
  byte t1;
  byte t2;

  thread Thread10 {
    loc loc0:
      when x != 0 do { t1 := x; }
      goto loc1;

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

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

```

```

thread Thread20 {
  loc loc0:
    when x != 0 do { t1 := x; }
    goto loc1;

```

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

```

thread Thread00 {
  loc loc0:
    do {
      x := 1;
    } goto loc1;

  loc loc1:
    do { assert (x != PARAM.N); }
    return;
}

```

# SumToN

```

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

  byte x;
  byte t1;
  byte t2;

  thread Thread10 {
    loc loc0:
      when x != 0 do { t1 := x; }
      goto loc1;

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

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

```

```

thread Thread20 {
  loc loc0:
    when x != 0 do { t1 := x; }
    goto loc1;

```

*declare three byte-sized variables*

```

thread Thread00 {
  loc loc0:
    do {
      x := 1;
    } goto loc1;

  loc loc1:
    do { assert (x != PARAM.N); }
    return;
}

```

# SumToN

```

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

  byte x;
  byte t1;
  byte t2;

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

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

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

```

```

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

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

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

return;
}

```

*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;
  byte t1;
  byte t2;

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

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

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

```

*The "main" thread initializes x, and then asserts that x is not equal to the value of N.*

```

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

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

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

thread Thread0() {
  loc loc0:
    do {
      x := 1;
    } goto loc1;

  loc loc1:
    do { assert (x != PARAM.N); }
    return;
}

```

# SumToN

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

  byte x;
  byte t1;
  byte t2;

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

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

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

**Note:** Arbitrary interleaving can occur between the execution of these two transitions

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

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

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

thread Thread0() {
  loc loc0:
    do {
      x := 1;
    } goto loc1;

  loc loc1:
    do { assert (x != PARAM.N); }
    return;
}
```

## Assessment

Can the assertion in the SumToN example be violated?

- 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

```

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

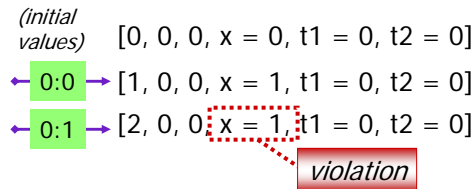
  loc loc1:
  k:1 do { t2 := x; }
      goto loc2;

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

thread Thread0() {
  loc loc0:
  0:0 do {
      x := 1;
    } goto loc1;

  loc loc1:
  0:1 do { assert (x != PARAM.N); }
      return;
}
    
```

## Violating schedule for $N = 1$



...that was easy!

# SumToN Assertion Violation

```

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

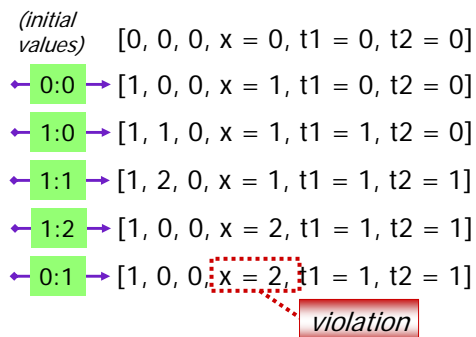
  loc loc1:
  k:1 do { t2 := x; }
      goto loc2;

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

thread Thread0() {
  loc loc0:
  0:0 do {
      x := 1;
    } goto loc1;

  loc loc1:
  0:1 do { assert (x != PARAM.N); }
      return;
}
    
```

## Violating schedule for $N = 2$



# SumToN Assertion Violation

```

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

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

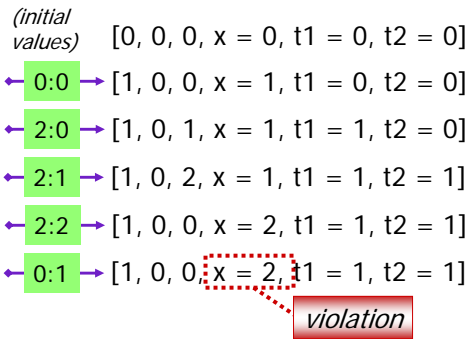
loc loc2:
do { x := t1 + t2; }
goto loc0;
}

thread Thread0() {
  loc loc0:
  do {
    x := 1;
  } goto loc1;
}

loc loc1:
do { assert (x != PARAM.N); }
return;
}
    
```

Another

Violating schedule for  $N = 2$



# SumToN Assertion Violation

```

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

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

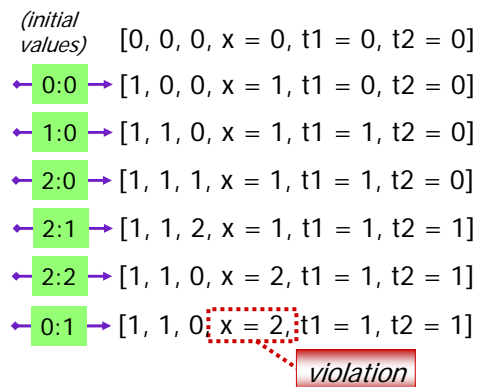
loc loc2:
do { x := t1 + t2; }
goto loc0;
}

thread Thread0() {
  loc loc0:
  do {
    x := 1;
  } goto loc1;
}

loc loc1:
do { assert (x != PARAM.N); }
return;
}
    
```

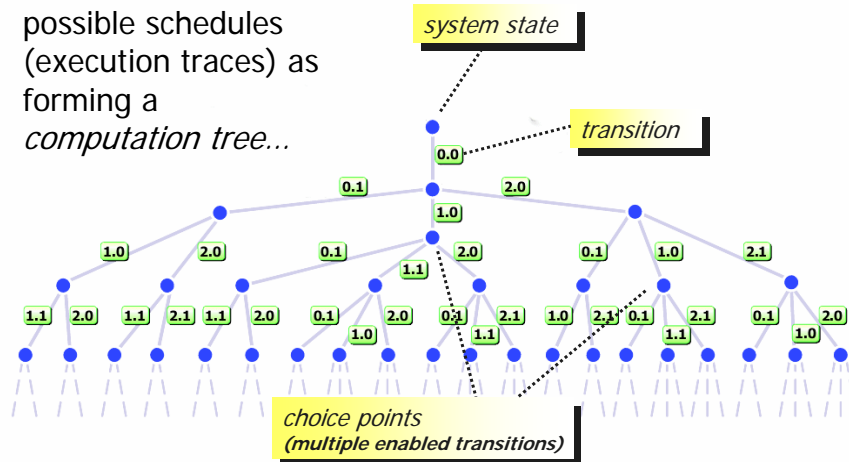
Yet Another

Violating schedule for  $N = 2$



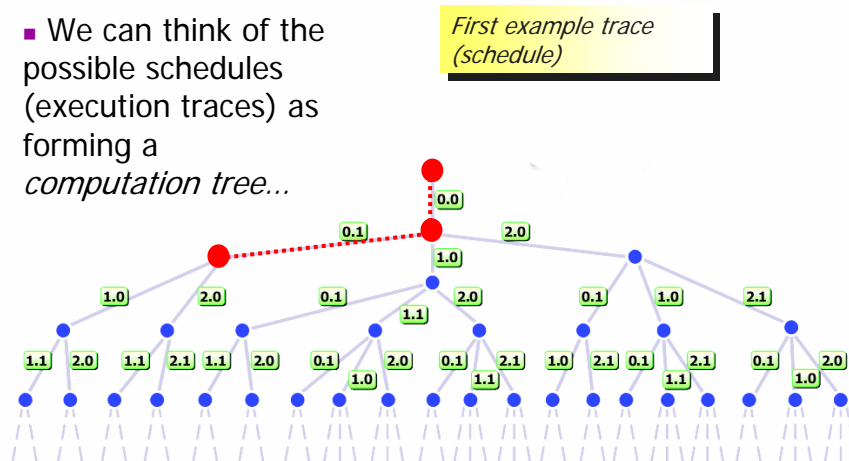
# Computation Tree

- 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*...







## Summary

- The actions/transitions of a concurrent system can be interleaved in arbitrary ways
- The implementation's *scheduler* is responsible for choosing a particular ordering of transitions
- When verifying systems, one often abstracts away from the particular scheduling strategy and considers the choices between enabled transitions to be made non-deterministically
- The collection of execution traces of a system can be visualized as a computation tree
- A path through the computation tree corresponds to a particular execution (i.e., schedule).