

# CIS 842: Specification and Verification of Reactive Systems

## Lecture INTRO-BIR-Lite: BIR-Lite Syntax and Semantics

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 basic syntactic structure of the BIR-Lite modeling language
- Understand how BIR-Lite systems can be described mathematically using the notion of *state transition system*.
- Be able to describe simple concurrent systems in BIR-Lite

*Be sure you have Bogor installed  
before you start this lecture!*

## Outline

- What is BIR and BIR-Lite
  - why are we using BIR-Lite?
- Syntax of BIR-Lite
  - illustrated by example
  - see book for grammar
- Semantics of BIR-Lite
  - BIR-Lite systems as state transition systems

## BIR

BIR = Bandera Intermediate Representation

- Used as the intermediate language for the Bandera Tool Set for model-checking Java programs
- Guarded command language
  - `when <condition> do <command>`
- Native support for a variety of object-oriented language features
  - dynamically created objects and threads, exceptions, methods, inheritance, etc.

...but we don't need all of this to learn the basics of model-checking

# BIR-Lite

## Subset of BIR

- Contains features that allow us to explore numerous issues related to modeling-checking in a simple and clean setting
- Does not contain the many features of BIR associated with OO software
  - omits methods, dynamically created data and threads, exceptions, etc.
- Has a straightforward correspondence with formal structures that are often used in the literature when describing foundational aspect of model-checking algorithms
- We will illustrate the syntax of BIR-Lite with an example
  - a grammar can be found in the appendix of the book.

# Simple Dining Philosophers

```
system TwoDiningPhilosophers {
  boolean fork1;
  boolean fork2;
  thread Philosopher1() {
    loc loc0: // take first fork
    when !fork1 do { fork1 := true; }
    goto loc1;

    loc loc1: // take second fork
    when !fork2 do { fork2 := true; }
    goto loc2;

    loc loc2: // put second fork
    do { fork2 := false; }
    goto loc3;

    loc loc3: // put first fork
    do { fork1 := false; }
    goto loc0;
  }
  thread Philosopher2() {
    loc loc0: // take second fork
    when !fork2 do { fork2 := true; }
    goto loc1;

    loc loc1: // take first fork
    when !fork1 do { fork1 := true; }
    goto loc2;

    loc loc2: // put first fork
    do { fork1 := false; }
    goto loc3;

    loc loc3: // put second fork
    do { fork2 := false; }
    goto loc0;
  }
}
```

*variable declaration*

# Simple Dining Philosophers

```
system TwoDiningPhilosophers {
  boolean fork1;
  boolean fork2;

  -----
  thread Philosopher1() {
    loc loc0: // take first fork
    when !fork1 do { fork1 := true; }
    goto loc1;

    loc loc1: // take second fork
    when !fork2 do { fork2 := true; }
    goto loc2;

    loc loc2: // put second fork
    do { fork2 := false; }
    goto loc3;

    loc loc3: // put first fork
    do { fork1 := false; }
    goto loc0;
  }
}
```

*thread declarations*

```
thread Philosopher2() {
  loc loc0: // take second fork
  when !fork2 do { fork2 := true; }
  goto loc1;

  loc loc1: // take first fork
  when !fork1 do { fork1 := true; }
  goto loc2;

  loc loc2: // put first fork
  do { fork1 := false; }
  goto loc3;

  loc loc3: // put second fork
  do { fork2 := false; }
  goto loc0;
}}
```

CIS 842: INTRO: BIR Syntax and Semantics

7

# Simple Dining Philosophers

*control  
locations*

```
system TwoDiningPhilosophers {
  boolean fork1;
  boolean fork2;

  thread Philosopher1() {
    loc loc0: // take first fork
    when !fork1 do { fork1 := true; }
    goto loc1;

    loc loc1: // take second fork
    when !fork2 do { fork2 := true; }
    goto loc2;

    loc loc2: // put second fork
    do { fork2 := false; }
    goto loc3;

    loc loc3: // put first fork
    do { fork1 := false; }
    goto loc0;
  }
}
```

CIS 842: INTRO: BIR Syntax and Semantics

8

# Simple Dining Philosophers

Guarded transformations

...aka "guarded transitions",  
"guarded commands"

when  
condition is  
true, ...

```
thread Philosopher1() {  
  loc loc0: // take first fork  
  when !fork1 do { fork1 := true; }  
  goto loc1;  
  
  loc loc1: // take second fork  
  when !fork2 do { fork2 := true; }  
  goto loc2;  
  
  loc loc2: // put second fork  
  do { fork2 := false; }  
  goto loc3;  
  
  loc loc3: // put first fork  
  do { fork1 := false; }  
  goto loc0;  
}
```

execute these  
statement(s)  
atomically

trivially true  
guard...

CIS 842: INTRO: BIR Syntax and Semantics

9

## For You To Do...

- Choose one of the simple concurrent systems from Chapter 1 and code that system in BIR-Lite
- Load your code (or edit it directly) in Bogor and experiment with Bogor's file open/save/close operations as well as Bogor's syntax checking and type checking facilities
  - See guided exercise in the Companion

CIS 842: INTRO: BIR Syntax and Semantics

10

## BIR-Lite Execution

- A BIR-Lite execution can be viewed as sequence of atomic steps or *transitions* between system states.
- Intuitively, a BIR-Lite state holds all information that is necessary to carry on computation starting from a particular point in the system's execution

## BIR-Lite State

A BIR-Lite state consists of...

```
boolean fork1;  
boolean fork2;
```

values of system variables

```
thread Philosopher1() {  
  loc loc0: // take first fork  
  when !fork1 do { fork1 := true; }  
  goto loc1;
```

```
  loc loc1: // take second fork  
  when !fork2 do { fork2 := true; }  
  goto loc2;
```

```
  loc loc2: // put second fork  
  do { fork2 := false; }  
  goto loc3;
```

```
  loc loc3: // put first fork  
  do { fork1 := false; }  
  goto loc0;  
}
```

current control location (program counter) of each thread

```
thread Philosopher2() {  
  loc loc0: // take second fork  
  when !fork2 do { fork2 := true; }  
  goto loc1;
```

```
  loc loc1: // take first fork  
  when !fork1 do { fork1 := true; }  
  goto loc2;
```

```
  loc loc2: // put first fork  
  do { fork1 := false; }  
  goto loc3;
```

```
  loc loc3: // put second fork  
  do { fork2 := false; }  
  goto loc0;  
}}
```

## Notation for BIR-Lite State

### System State

```
[pc1 -> 0,  
pc2 -> 1,  
fork1 -> false,  
fork2 -> true]
```

*...pc for Philosopher1 is loc0*  
*...pc for Philosopher2 is loc1*  
*...value of fork1 is 'false'*  
*...value of fork2 is 'true'*

*...sometimes we will abbreviate as...*

```
[0, 1, false, true]
```

*...if the ordering of variable values is clear  
from the context*

## BIR-Lite Transition Notation

```
[pc1 ↦ 2, pc2 ↦ 0, fork1 ↦ "true", fork2 ↦ "true"]  
1:2  
→ [pc1 ↦ 3, pc2 ↦ 0, fork1 ↦ "true", fork2 ↦ "false"]
```

*The thread Philosopher<sub>1</sub> executes the  
transition leading out of loc<sub>2</sub>*

```
thread Philosopher1() {  
  ...  
  loc loc2: // put second fork  
  do { fork2 := false; }  
  goto loc3;  
  
  loc loc3: // put first fork  
  do { fork1 := false; }  
  goto loc0;  
}
```

# BIR-Lite Execution Trace

An execution trace is a sequence of transitions between states

```
      [pc1 ↦ 0, pc2 ↦ 0, fork1 ↦ "false", fork2 ↦ "false"]
1:0  ↦ [pc1 ↦ 1, pc2 ↦ 0, fork1 ↦ "true", fork2 ↦ "false"]
1:1  ↦ [pc1 ↦ 2, pc2 ↦ 0, fork1 ↦ "true", fork2 ↦ "true"]
1:2  ↦ [pc1 ↦ 3, pc2 ↦ 0, fork1 ↦ "true", fork2 ↦ "false"]
2:0  ↦ [pc1 ↦ 3, pc2 ↦ 1, fork1 ↦ "true", fork2 ↦ "true"]
1:3  ↦ [pc1 ↦ 0, pc2 ↦ 1, fork1 ↦ "false", fork2 ↦ "true"]
2:1  ↦ [pc1 ↦ 0, pc2 ↦ 2, fork1 ↦ "true", fork2 ↦ "true"]
2:2  ↦ [pc1 ↦ 0, pc2 ↦ 3, fork1 ↦ "false", fork2 ↦ "true"]
2:3  ↦ [pc1 ↦ 0, pc2 ↦ 0, fork1 ↦ "false", fork2 ↦ "true"]
→    ...
```

...with these notions in place, we can consider a more formal presentation of simple concurrent systems

# State Transition System

A *state transition system* is a mathematical structure that will allow us to reason precisely about the behavior of simple concurrent systems.

$$\Sigma = (S, T, S_0, L)$$

...describes the behavior of a particular system

- $S$  is a set of *system states*
- $T$  is a set of *transitions*
  - each transition  $\delta$  is a relation between states
- $S_0$  is a set of *initial states*
- $L$  maps a state  $s$  to a set of primitive properties that are true of  $s$  (*we won't use  $L$  for a while*).

## Explanation By Example

*...we now explain the correspondence between a BIR-Lite system and its associated state transition system using the Dining Philosophers example – we refer to the associated state transition system as  $\Sigma_{2DP}$ .*

## States

### $\Sigma_{2DP}$ states

- There are 64 states in S
  - 4 locations for Philosopher1
  - 4 locations for Philosopher2
  - 2 values for fork1
  - 2 values for fork2

## Initial States

### $\Sigma_{2DP}$ initial states

- BIR-Lite systems always have exactly one initial state
  - program counters for each thread are set to initial location (first location in thread code)
  - each BIR type has a default value
- In  $\Sigma_{2DP}$   $S_0 = [0, 0, \text{false}, \text{false}]$

## Transitions

### $\Sigma_{2DP}$ transitions

- For an arbitrary BIR-Lite system, there will be a transition in  $T$  for each guarded transformation in the BIR-Lite code.
- In  $\Sigma_{2DP}$ , there are eight transitions in  $T$  – one for each of the four transitions in each of the four threads.
- Each transition leading out of BIR location  $L$  in thread  $t$  has an implicit guard that only allows it to fire when  $t$ 's program counter is at  $L$ .

## Transitions as Relations/Functions

- Formally,  $\delta$  is a subset of  $S \times S$ , i.e.,
  - $\delta$  is a relation between states
  - given an input state  $s$ ,  $\delta(s)$  yields the states output by the transition.
- We will only need to consider *deterministic* transitions, i.e.,
  - transitions that are (partial) functions, i.e.,
  - transitions that, given an input state are *undefined* or produce *exactly one output state*.

## Transition Examples

```
thread Philosopher1() {
  loc loc0: // take first fork
  when !fork1 do { fork1 := true; }
  goto loc1;
```

```
loc loc1: // take second fork
  when !fork2 do { fork2 := true; }
  goto loc2;
```

```
loc loc2: // put second fork
  do { fork2 := false; }
  goto loc3;
```

```
loc loc3: // put first fork
  do { fork1 := false; }
```

*...move to next control location, and execute assignments in transformation.*

$$\alpha_{1:1}([1, 0, \text{"true"}, \text{"false"}]) = [2, 0, \text{"true"}, \text{"true"}]$$

$$\alpha_{1:1}([1, 2, \text{"false"}, \text{"false"}]) = [2, 2, \text{"false"}, \text{"true"}]$$

## Enabled/Disabled Transitions

- A BIR-Lite transformation is *enabled* for a particular state if both its explicit guard (on data variables) and implicit guard (on the current control position) are true.
- Relating to transitions from  $\Sigma$ ,
  - a transition  $\delta$  is *enabled* for a state  $s$  if there exists a state  $s'$  such that  $\delta(s) = s'$  (i.e.,  $\delta$  is *defined* on  $s$ )
  - a transition  $\delta$  is *disabled* for a state  $s$  if  $\delta$  is undefined on  $s$ .

## Enabled/Disabled Transitions

### Examples

...the transition  $\delta_{1;1}$  is undefined (disabled) on each of the states below

```
[1, 1, "true", "true"]  
[0, 0, "false", "false"]  
[1, 2, "false", "true"]
```

```
thread Philosopher1() {  
  loc loc0: // take first fork  
  when !fork1 do { fork1 := true; }  
  goto loc1;  
  
  loc loc1: // take second fork  
  when !fork2 do { fork2 := true; }  
  goto loc2;  
  
  ...  
}
```

...because fork2 is *true*  
...because  $pc_1$  is not loc1  
...because fork2 is *true*

## Enabled/Disabled Notation

- $enabled(s)$ 
  - the set of transitions enabled at  $s$
- $enabled(s,t)$ 
  - the set of transitions from thread  $t$  that are enabled at  $s$
- $disabled(s), disabled(s,t)$ 
  - ...similar to above
- $pc(s,t)$ 
  - the value in  $s$  of the program counter for  $t$
- $current(s), current(s,t)$ 
  - the set of all transitions (whether enabled or disabled) associated with the current program counters of  $s$ .

## Enabled/Disabled Transitions

```

thread Philosopher1() {
  loc loc0: // take first fork
  when !fork1 do { fork1 := true; }
  goto loc1;

  loc loc1: // take second fork
  when !fork2 do { fork2 := true; }
  goto loc2;

  loc loc2: // put second fork
  do { fork2 := false; }
  goto loc3;

  loc loc3: // put first fork
  do { fork1 := false; }
  goto loc0;
}
    
```

```

thread Philosopher2() {
  loc loc0: // take second fork
  when !fork2 do { fork2 := true; }
  goto loc1;

  loc loc1: // take first fork
  when !fork1 do { fork1 := true; }
  goto loc2;

  loc loc2: // put first fork
  do { fork1 := false; }
  goto loc3;

  loc loc3: // put second fork
  do { fork2 := false; }
  goto loc0;
}}
    
```

$$\begin{aligned}
 enabled([0, 0, "false", "false"]) &= \{\alpha_{1:0}, \alpha_{2:0}\} \\
 enabled([1, 0, "true", "false"]) &= \{\alpha_{1:1}, \alpha_{2:0}\} \\
 enabled([2, 0, "true", "true"]) &= \{\alpha_{1:2}\} \\
 enabled([1, 1, "true", "true"]) &= \emptyset \\
 enabled([1, 0, "true", "false"], 1) &= \{\alpha_{1:1}\} \\
 enabled([2, 0, "true", "true"], 2) &= \emptyset
 \end{aligned}$$

## Current Transitions

```
thread Philosopher1() {
  loc loc0: // take first fork
  when !fork1 do { fork1 := true; }
  goto loc1;

  loc loc1: // take second fork
  when !fork2 do { fork2 := true; }
  goto loc2;

  loc loc2: // put second fork
  do { fork2 := false; }
  goto loc3;

  loc loc3: // put first fork
  do { fork1 := false; }
  goto loc0;
}

thread Philosopher2() {
  loc loc0: // take second fork
  when !fork2 do { fork2 := true; }
  goto loc1;

  loc loc1: // take first fork
  when !fork1 do { fork1 := true; }
  goto loc2;

  loc loc2: // put first fork
  do { fork1 := false; }
  goto loc3;

  loc loc3: // put second fork
  do { fork2 := false; }
  goto loc0;
}}
```

$$\begin{aligned} pc([0, 0, \text{"false"}, \text{"false"}], 1) &= 0 \\ pc([0, 2, \text{"true"}, \text{"true"}], 2) &= 2 \\ current([2, 0, \text{"true"}, \text{"true"}]) &= \{\alpha_{1:2}, \alpha_{2:0}\} \end{aligned}$$

## Reachable States

Often, we will only want to talk about the set of states that are reachable from the initial system state

- The following state is unreachable by starting with the initial state from  $S_0$  and applying transitions in  $T$

$$[pc_1 \mapsto 2, pc_2 \mapsto 0, fork1 \mapsto \text{"false"}, fork2 \mapsto \text{"false"}]$$

- if Philosopher1 has reached loc2, it will have already acquired both forks (i.e., the forks should have the value *true*)

## Reachable States

When we talk about a system's "state space", we usually mean its "reachable state space"

- The reachable state space of a concurrent system is one interesting measure of its complexity
- How many reachable states does our dining philosopher system have?

## Non-determinism

How does system execution proceed at a state  $s$  when there is more than one transition in  $enabled(s)$ ?

- In a real implementation, the question above is resolved by the run-time system's *scheduler*
- When reasoning about system models, one typically considers the choice of transition to execute from  $enabled(s)$  to be *non-deterministic*.
  - this abstracts away from a particular scheduling strategy
- In BIR-Lite systems, there are two sources of non-determinism
  - intra-thread non-determinism
    - multiply enabled transitions in the same thread
  - inter-thread non-determinism
    - multiply enabled transitions from different threads

## Intra-Thread Non-determinism

Sometimes there can be more than one enabled transition for a particular thread

```
loc loc0:  
  when A do x := x + 1 goto loc1;  
  when B do y := y + 1 goto loc2;
```

*...when both A and B are true, one of the transitions leading out of loc0 is non-deterministically chosen for execution*

## For You Do To...

- Consider the X BIR-Lite example from the Book and the corresponding state transition system
  - How many transitions are in the system?
  - What is the size of the state-space for the system?
  - Give an estimate of the size of the reachable state-space of the system, and justify your reasoning.
  - Using the system, give examples of
    - five reachable states
    - three enabled transitions
    - five current transitions
    - a current transition that is not enabled
    - a prefix of an execution of the system that is at least seven transitions long

## Summary

- BIR-Lite is a simple modeling language for concurrent systems that we will use as a pedagogical vehicle for studying many aspects of model-checking.