

# CIS 842: Specification and Verification of Reactive Systems

## Lecture INTRO-Examples: Simple BIR-Lite Examples

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 how Bogor can be used to checking properties of simple examples of concurrent systems written in BIR-Lite.
- Understand basic correctness conditions for mutual exclusion protocols.

## Outline

- Implementing invariant checking using assertions in Bogor
- Simple invariant properties of sequential programs
- Checking invariants for concurrent programs
- Properties to check for mutual exclusion protocols
- Simple mutual exclusion examples

## Multiplication Example

Consider a simple program with a loop invariant

```
// assume parameters m and n
count := m;
output := 0;

// loop invariant: m * n == output + (count * n)
while (count > 0) do {
  output := output + n;
  count := count - 1;
}
```

*...let's write this in BIR*

# Multiplication Example

## BIR Version

*...using two threads is unnatural, but the motivation will be clear in a moment...*

```
system Mult {
  int m;
  int n;
  int count;
  int output;

  main thread Main () {
    loc loc0:
    do {m := (int (0,255)) 5;
      n := (int (0,255)) 4;
      count := m;
      output := (int (0,255)) 0;
      start T1();
    } return;
  }
}
```

```
thread T1 () {
  loc loc0:
  when (count > 0)
    do {output := output + n;
      count := count - 1;}
  goto loc0;
  when (count == 0)
    do {}
  return;
}
```

*Note: no interleaving between these two assignments*

*...how to program the check of the invariant?*

# Checking Invariants

- There are usually several ways to check an invariant / in most model checkers...
  - Temporal logic:
    - $[[I]]$
    - Built in specification form:
      - `invariant I`
  - But there is another way that takes advantage of the fact that a model-checker explores all interleavings of threads
    - *...and this can also further exercise your intuition about a model-checker's search behavior...*

# Checking Invariants

For a program with threads *Main, T1, ..., Tn*, where an invariant *I* is to be enforced on the actions of *Ti*, add an assertion of *I* as the last transition of Main

```
main thread Main ()
...
...
loc locAssert:
do {assert (I);}
return;
```

Due to the fact that the model-checker will explore all possible interleavings between Main and the other threads, the assertion statement will get interleaved (on some trace) between each execution steps of the other threads – thus, checking the invariant on every state produced by the execution of *T1, ..., Tn*

# Multiplication Example

## BIR Version

```
system Mult {
...
main thread Main () {
loc loc0:
do {m := (int (0,255)) 5;
n := (int (0,255)) 4;
count := m;
output := (int (0,255)) 0;
start T1();
} goto loc1;

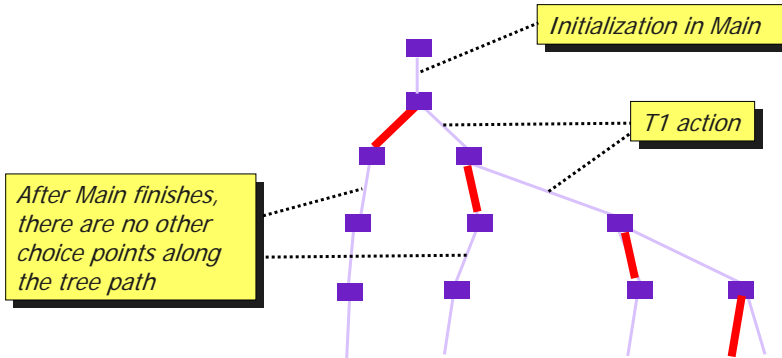
loc loc1:
do {assert (m*n ==
output+(count*n));}
return;
}
```

```
thread T1 () {
loc loc0:
when (count > 0)
do {output := output + n;
count := count - 1;}
goto loc0;
when (count == 0)
do {}
return;
}
```

Assertion added

## Computation Tree

— assertion transition (loc1 in Main)



In other words, there exists a path where we do 0 steps of T1 then check I, there exists a path where we do 1 step of T1 then check I, there exists a path where we do 2 steps of T1, then check I, etc.

CIS 842: INTRO: Simple BIR-Lite Examples

9

## Global Invariants

- When writing a concurrent program, the notion of *global invariant* has long been used as a mechanism for design the program.
- In fact, a tool developed in the SAnToS lab called *SyncGen* automatically synthesizes programs from global invariants  
(<http://syncgen.projects.cis.ksu.edu>)

CIS 842: INTRO: Simple BIR-Lite Examples

10

## Readers Writers Problem

- Considers a shared buffer being accessed in a system with multiple readers and writers, where we can allow multiple simultaneous read accesses.
- Let  $nr$  represent the number of readers in their critical regions accessing the buffer and let  $nw$  represent the number of writers in their critical regions accessing the buffer.
  - the associated global invariant is...
    - $nw \leq 1$  (never more than 1 writer at a time) and
    - $nr == 0$  or  $nw == 0$  (it must always be the case that either there are no readers in their critical sections, or no writers in their critical sections).

CIS 842: INTRO: Simple BIR-Lite Examples

11

## Readers/Writers Problem

```
system ReadersWriter
{ const PARAM
  { NUM_READERS = 2;
    NUM_WRITERS = 2;}

  int (0, PARAM.NUM_READERS) nr;
  int (0, PARAM.NUM_WRITERS) nw;

  thread Reader[NUM_READERS]() {
    loc loc0:
      when nw == 0 do {
        nr := nr + 1;} goto loc1;

    loc loc1:
      do { // read data
        } goto loc2;

    loc loc2:
      do { nr := nr - 1;} goto loc0;
  }

  thread Writer[NUM_WRITERS]() {
    loc loc0:
      when nw == 0 && nr == 0 do {
        nw := nw + 1;} goto loc1;

    loc loc1:
      do { // write data
        } goto loc2;

    loc loc2:
      do {nw := nw - 1;} goto loc0;
  }

  main thread Main() {
    // ... start threads ...
    loc loc0:
      do {assert (
        (nw <= 1) &&
        ((nr == 0) || (nw = 0)));}
      return;
  }
}
```

CIS 842: INTRO: Simple BIR-Lite Examples

12

## Mutual Exclusion Protocols

A good solution to the mutual-exclusion problem is expected to satisfy the following two requirements:

- *Exclusion*: No computation path should include a state where both processes are executing in their critical regions at the same time
- *Accessibility*: Every state in a computation path in which one process is in a state immediately before its critical region must eventually be followed by another state in which the process is in its critical section.
- *Note*: together these requirements imply that once a process enters a critical region, it should always exit it.

## Mutual Exclusion Protocols

First attempt:

- For two threads, have two flags `flag1` and `flag2` where each flag indicates if a thread has entered its critical region.

# Mutual Exclusion Protocols

```
system MuxTry {
  boolean flag1;
  boolean flag2;

  thread T1 () {
    loc loc0: live {}
    do {flag1 := true;} goto loc2;

    loc loc2: live {}
    when (!flag2)
    do {} goto loc3;

    loc loc3: live {}
    do {} goto loc4;

    loc loc4: live {}
    do {flag1 := false;} goto loc0;
  }

  thread T2 () {
    loc loc0: live {}
    do {flag2 := true;} goto loc2;

    loc loc2: live {}
    when (!flag1)
    do {} goto loc3;

    loc loc3: live {}
    do {} goto loc4;

    loc loc4: live {}
    do {flag2 := false;} goto loc0;
  }
}
```

*critical regions*

CIS 842: INTRO: Simple BIR-Lite Examples

source: Manna & Pnueli

15

## For You To Do...

- Does the system on the previous page satisfy the accessibility property?
  - use Bogor to check this
- Does the system on the previous page satisfy the exclusion property?
  - how can you check the exclusion property in Bogor?

CIS 842: INTRO: Simple BIR-Lite Examples

16

## Assessment

- Running Bogor on the MuxTry program reveals that *Accessibility* is *not* satisfied...
  - both threads may set their flags to true, and then block before either thread actually enters the critical section.
- For *Exclusion*, we need to instrument the program with a counter variable to count the number of threads in the critical region, and then use the trick of putting an assertion as the last step of the main method to state an invariant that the counter never has the value of 2 (we will see the code for this later).

## Assessment

- A solution to the problem of MuxTry is to have a “tie-breaking” variable that indicates that if both threads are waiting to enter, then a certain thread has priority.
- Introduce a new variable **turn** that indicates the thread *whose turn it is to wait* while the other thread enters the critical region.
- This solution is due to Peterson **Reference!**

# Peterson's Solution

```

system Peterson {
  boolean flag1;
  boolean flag2;
  Int (1,2) turn;

  thread T1 () {
    loc loc0: live {}
    do {flag1 := true;
      turn := (Int (1,2)) 1;
    } goto loc2

    do {loc loc2: live {}
      when (!flag2 ||
        (turn == (Int (1,2)) 2))
      do {} goto loc3;

      loc loc3: live {}
      do {} goto loc4;

      loc loc4: live {}
      do {flag1 := false;} goto loc1;
    }
  }

  thread T2 () {
    loc loc0: live {}
    do {flag2 := true;
      turn := (Int (1,2)) 2;
    } goto loc2;

    loc loc2: live {}
    when (!flag1 ||
      (turn == (Int (1,2)) 1))
    do {} goto loc3;

    loc loc3: live {}
    do {} goto loc4;

    loc loc4: live {}
    do {flag2 := false;} goto loc1;
  }
}

```

# For You To Do...

- Does the system on the previous page satisfy the accessibility property?
  - use Bogor to check this
- Does the system on the previous page satisfy the exclusion property?
  - use Bogor to check this

## Assessment

```
thread T2 () {
  loc loc0: live {}
  do {flag2 := true;
     turn := (Int (1,2)) 2;
  } goto loc2;

  loc loc2: live {}
  when (!flag1 ||
        (turn == (Int (1,2)) 2))
  do {} goto loc3;

  loc loc3: live {}
  do {} goto loc4;

  loc loc4: live {}
  do {flag2 := false; } goto loc1;
}
```

- Consider if we were checking a solution to be implemented at the machine instruction level...
- The granularity of our instruction steps is too coarse...
  - You can't do two assignments (or two tests) in the same instruction.

## For You To Do...

- Modify the Peterson solution so that each BIR transition corresponds to one machine instruction, and re-check the correctness properties of the algorithm.
- Does the system still satisfy both properties?

## Assessment

- Often times when we are modeling systems, we make choices about the granularity of instructions.
- This can affect the costs of model-checking...
  - larger granularity means fewer states and interleavings
- This can affect verification...
  - making the granularity “too large” can mask some interleavings that could lead to property violations
- Therefore, the choice of instruction granularity is a subtle but important issue when modeling systems and model-checking

## Peterson's Solution -- Revisited

- Perhaps we have not considered the full implications of the *Accessibility* requirement...
  - Does there exist a computation path where T1 is sitting ready to enter its critical region but T2 keeps going around its loop and T1 never gets to go?
  - Yes, this is a liveness violation which requires a *nested* depth-first search, which we have not implemented yet. The nested search checks for cycles.
  - This is also associated with *fairness* issues – the trace described above would be unfair to T1 because if though it is enabled infinitely often, it never gets to move.
- We will discuss all these issues later in the course.

## Summary

- We have seen how BIR can be used to model a number of simple systems.
- We've continued to reveal how programming very simple concurrent systems can be quite tricky.
- Liveness and Fairness are other important issues that we will need to explore later in the course.